INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI[®]

Performance Evaluation of Query Processing on Network of Workstations

by

Shi-Min Qian

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements
for the degree of Master of Computer Science

Ottawa-Carleton Institute for Computer Science

School of Computer Science

Carleton University

Ottawa, Ontario

September, 1999

© 1999, Shi-Min Qian

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-48440-8

Canada

The undersigned hereby recommend to
the Faculty of Graduate Studies and Research
acceptance of the thesis,

**Performance Evaluation of Query Processing
on Network of Workstations**

submitted by

Shi-Min Qian, B. Eng., M. Sc.

in partial fulfillment of the requirements
for the degree of Master of Computer Science

A handwritten signature in dark ink, appearing to read 'E. Grimalis', is written over a horizontal line.

Director, School of Computer Science

A handwritten signature in dark ink, consisting of several loops and a long horizontal stroke, is written over a horizontal line.

Thesis Supervisor

Carleton University

September 14, 1999

Abstract

Network of Workstations (NOW) has increasingly become a viable, flexible yet inexpensive platform for parallel/distributed computing. Database query processing is one such area that can benefit from the use of this platform. In this thesis, a general overview of the NOW research is presented. A study has been conducted to quantify the performance difference of three predominant architectures in NOW environment, namely, centralized, distributed and hierarchical architectures, to process queries directed to a centralized database. Simulations have been performed to simulate the environment of multiple computers (PC or workstations) interconnected with a centralized database via a general-purpose communication network such as LAN. Results indicate that parallel query processing utilizing NOW exhibits good scaleup and speedup. Most importantly, all three architectures demonstrate performance sensitivity to fragmentation size. With the ideal fragmentation size used in all three architectures, however, hierarchical architecture consistently outperforms centralized and distributed architectures.

To Hui

Acknowledgements

First and foremost, I thank my thesis supervisor, Dr. Sivarama Dandamudi for his guidance and encouragement since my early days in Computer Science. This thesis work would not be possible without his inspiration and advice. He offers students fairness and understanding, which I was lucky enough to experience under his supervision. I am grateful to Dr. John Oommen for I have benefited greatly from his generous assistance and good faith. Many thanks to Dr. Nel and Dr. Raymond for providing valuable suggestions and comments on the thesis.

Thanks go to our retired graduate secretary Marlene Wilson for her prompt administrative support and Andrew Miles for answering my computer access and account questions. I enjoyed those long hours working with my fellow students on the course projects. I thank Carleton University for accommodating students like me who have a full-time career in parallel with the in-program study. I wish to thank my manager at Nortel, Murat Ozmizrak, who generously agreed to offer educational assistance.

I am indebted to my parents, who lent their helping hands while I was in the final stage of my thesis work. Without them, I could not have accomplished all these in such a short period. I am also thankful for having a grandmother whose unconditional love for me is and will always be my most cherished gift and inspiration in life. Last, but not the least, this is for you – Hui, for your patience, caring, and insights. My heartfelt thanks to you for cheerfully sharing this period of my life when I struggled between career and study, and most of all, for lending me your shoulders all these years whenever I needed.

Table of Contents

ABSTRACT.....	III
ACKNOWLEDGEMENTS	V
TABLE OF CONTENTS	VI
LIST OF FIGURES.....	IX
1. INTRODUCTION.....	1
1.1 PARALLEL QUERY PROCESSING	4
<i>1.1.1 Query Processing Concepts.....</i>	<i>4</i>
<i>1.1.2 Basic Types of Parallelism.....</i>	<i>5</i>
<i>1.1.3 Performance of Parallel Databases.....</i>	<i>8</i>
1.2 NETWORK OF WORKSTATIONS	11
<i>1.2.1 Historical Development</i>	<i>12</i>
<i>1.2.2 Issues.....</i>	<i>14</i>
1.3 MOTIVATION.....	15
1.4 OVERVIEW OF THE THESIS.....	16
2. BACKGROUND	17

2.1 SCHEDULING	17
2.1.1 <i>Load Balancing and Load Sharing</i>	18
2.1.2 <i>Processor Scheduling Policies</i>	24
2.2 HIGH PERFORMANCE NETWORKING OF NOW	28
2.2.1 <i>Switch-based Networks</i>	29
2.2.1.1 Myrinet.....	31
2.2.1.2 ATM.....	32
2.2.2 <i>Communication Protocols</i>	32
2.3 QUERY PROCESSING STRATEGIES	37
2.3.1 <i>Query Processing of Parallel Databases</i>	37
2.3.1.1 Parallel Join Algorithms	37
I Parallel nested-loop join	37
II Parallel sort-merge join.....	38
III Parallel hash-based join	39
2.3.1.2 Architectures of Parallel Database Systems.....	41
I Shared-Disk Architecture.....	41
II Shared-Nothing Architecture	45
2.3.2 <i>Query Processing on NOW</i>	48
3. CENTRALIZED, DISTRIBUTED AND HIERARCHICAL	
ARCHITECTURES.....	51
3.1 CENTRALIZED ARCHITECTURE	53

3.1.1	<i>Overview</i>	53
3.1.2	<i>Simulation Model</i>	56
3.2	DISTRIBUTED ARCHITECTURE	62
3.2.1	<i>Overview</i>	62
3.2.2	<i>Simulation Model</i>	65
3.3	HIERARCHICAL ARCHITECTURE	69
3.3.1	<i>Overview</i>	69
3.3.2	<i>Simulation Model</i>	71
3.4	NETWORK MODEL	79
4.	PERFORMANCE EVALUATION	81
4.1	SELECTION OF PARAMETERS	81
4.2	SYSTEM SCALEUP	82
4.3	SYSTEM SPEEDUP	85
4.4	PERFORMANCE SENSITIVITY TO FRAGMENTATION SIZES	86
4.5	PERFORMANCE AS A FUNCTION OF QUERY ARRIVAL RATE	96
4.6	EFFECT OF BACKGROUND LOAD ON PERFORMANCE	102
5.	CONCLUSIONS AND FUTURE WORK	105
5.1	SUMMARY	105
5.2	CONTRIBUTION OF THE THESIS	107
5.3	FUTURE WORK	107
	REFERENCES	109

List of Figure

Figure 1 Network of Workstations (NOW)	2
Figure 2 Query Processing on Network of Workstations (NOW)	3
Figure 3 Basic types of parallelism (Dewitt and Gray 1992).....	7
Figure 4 Shared-Disk Architecture	42
Figure 5 Shared-Nothing Architecture.....	46
Figure 6 Components of Query Processing on Network of Workstations (NOW).....	52
Figure 7 Centralized Task Queue Organization.....	54
Figure 8 Data flow in centralized architecture	57
Figure 9 Balanced query format.....	58
Figure 10 Control flow diagram of master process in centralized architecture	60
Figure 11 Control flow diagram of the ith slave process in centralized architecture.....	61
Figure 12 Distributed Run Queue Organization	63
Figure 13 Data flow in distributed architecture	66
Figure 14 Control flow diagram of the ith master process in distributed architecture	67
Figure 15 Control flow diagram of the ith slave process in distributed architecture.....	68
Figure 16 Hierarchical Run Queue Organization.....	70
Figure 17 Job transfer process in hierarchical run queue organization.....	72
Figure 18 Hierarchical task queues for query processing with processing node 0, ..., 7 ..	73

Figure 19 Query transfer mode, operation transfer mode and partial operation transfer mode in a hierarchical system of 64 nodes	75
Figure 20 Control flow diagram of master process in hierarchical architecture	77
Figure 21 Control flow diagram of the <i>i</i> th slave process in hierarchical architecture	78
Figure 22 System Scaleup under light system load.....	83
Figure 23 System Scaleup under heavy system load	84
Figure 24 System response time with fixed system workload (I. low arrival rate).....	87
Figure 25 System response time with fixed system load (II. high arrival rate).....	88
Figure 26 Speedup for 4x system (16 nodes vs 4 nodes, centralized, distributed and hierarchical)	89
Figure 27 Speedup for 16x system (64 nodes vs 4 nodes centralized and distributed) ...	90
Figure 28 Response time is sensitive to the fragmentation size (I. low arrival rate)	92
Figure 29 Response time is sensitive to the fragmentation size (II. high arrival rate).....	93
Figure 30 Response time is sensitive to the fragmentation size (III. 10Mbps)	94
Figure 31 Response time is sensitive to the fragmentation size in hierarchical architecture (a. light load; b. heavy load).....	95
Figure 32 Performance comparison of centralized, distributed and hierarchical architectures (1000Mbps)	99
Figure 33 Performance comparison of centralized, distributed and hierarchical architectures (100 Mbps)	99
Figure 34 Performance comparison of centralized, distributed and hierarchical architectures (10 Mbps)	101

Figure 35 The impact of periodic background load on response time	104
---	-----

Chapter 1

Introduction

With the popularity of PCs and the production of increasingly powerful workstations, parallel and distributed computer systems composed of multiple processors or multiple computers connected by high-speed bus or communication networks have been accepted as the way to achieve more processing power in many areas of computer applications. A major area of such application has been the information system that involves large databases with the facility to process complex queries and transaction tasks. This thesis is concerned with the performance of database query processing in a distributed computing environment, more specifically, Network of Workstations (NOW) environment (Figure 1 and Figure 2). A general introduction is presented in this chapter to this new class of parallel and distributed platform that consists of a cluster of otherwise idle workstations interconnected via high-speed communication networks such as local area networks (LAN). We will also briefly discuss relational database model, its

Figure 1 Network of Workstations (NOW)

Figure 2 Query Processing on Network of Workstations (NOW)

natural affinity to parallel execution and a variety of parallelisms that help databases speedup their query processing tasks. A number of query processing strategies discussed herein are employed in our performance study.

1.1 Parallel Query Processing

As this thesis focuses on database applications in a NOW environment, this section will provide a review of the basic database query processing concepts, the development of parallel database, and common types of parallelism employed by parallel systems.

1.1.1 Query Processing Concepts

The success of relational database model has provided the motivation for the exploitation of parallel and distributed architectures to improve the performance for very large database systems. This section will give a brief introduction of the basic concepts of relational database model and major operations that are performed on the data stored in relational databases.

A relational database consists of relations (correspond to files). Each relation is, in turn, composed of many tuples (correspond to records). All the tuples in a relation have the same set of attributes (correspond to fields).

Relations are created, updated and queried by writing SQL statements, which contain a set of operators evolved from relational algebra. One of the two most commonly used operations is select-project, which reads each tuple t of relation R and pick the tuples that satisfy the predicate P supplied (selection) and output the attribute values of those selected tuples that are specified in attribute list L (projection). Join is another important operation generally needed to answer multirelation queries, but it can be very time consuming. There are several types of join, all of which will produce a subset of the cross product of two relations based on finding all the matching tuples in A and B on the join predicate. Different types of join employ different types of join conditions (or join predicates). The most commonly used join is equi-join. The join condition of equi-join is characterized by equal sign " $=$ ". For instance, for each tuple a in relation A , the equi-join finds all tuples in relation B that have the attribute values equal to that of tuple a in A . For each matching pair of tuples, the natural join operator (a common type of equi-join) concatenates the pair and inserts it into the output stream.

Several implementations of SQL support many other sets of operators, such as aggregate operators (sum, min, max), usual set operators (union, intersection, difference) and some exotic ones like division. Since join operation that involves multiple relations is the most commonly used and most expensive operation in relational databases, the emphasis of our study will be on this operation.

1.1.2 Basic Types of Parallelism

Relational databases are known to have a certain natural affinity to parallelism. Relational queries are uniform set of operators applied to large collections of uniform data, which allow them to be arbitrarily composed into many possible dataflow graphs, offering opportunities for parallelism. The examples are pipelined parallelism and partitioned parallelism (Figure 3) that will be discussed in this section. Moreover, the essence of SQL statements lies in that they are written in a nonprocedural language. They only specify what to retrieve, but not how to implement [24]. This feature itself would leave considerable latitude in the ways the query can be executed. The types of parallelisms can be categorized into two basic classes based on the particular level at which the parallelism can be exploited: inter-query parallelism and intraquery parallelism. Intraquery parallelism can be further divided into interoperator parallelism and intraoperator parallelism. Extensive studies on intraquery parallelism and inter-query parallelism have shown that both methods can improve query execution time (for review see [24]). For interquery parallelism, several queries are executed in parallel. For intraquery parallelism, a query optimizer produces a query plan that could be

Pipeline parallelism [24]

Partitioned data allows data parallelism [24]

Figure 3 Basic types of parallelism

executed in parallel by a number of processors. Interoperator parallelism means several operators within a query can be executed in parallel, whereas intraoperator parallelism parallelizes the steps within each operator, i.e., parallelism in each operator occurs in such a way that multiple processors work in parallel on a single database operation.

Distributed databases and parallel databases also employ two other basic (orthogonal) mechanisms of parallelism called program parallelism (PP) and data parallelism (DP) [52]. Pipelined parallelism is an example of program parallelism and data parallelism is typified by data partitioning (Figure 3). By streaming the output of one operator into the input of another operator, two operators that are assigned to two different processors can work concurrently. This constitutes pipelined parallelism at the interoperator level. Pipelining can also be applied to steps within an operator. By partitioning input data among multiple processors so that each processor works concurrently on a part of the data, data parallelism can be achieved.

1.1.3 Performance of Parallel Databases

Originally, the support and maintenance of very large databases (VLDB) were envisioned to utilize specialized computers termed as “database machines” [24]. In 1970s, there were attempts to design specialized database machine by exploiting specialized hardware such as CCD memories, bubble memories and optical disks. In the end, none of these

technologies delivered the performance as expected [24]. During the past two decades, however, two major factors came into play. One is the phasing out of mainframe computers, replaced by smaller, cheaper, yet increasingly powerful PCs and workstations. This trend inspired interests in using these computing facilities in a parallel and distributed fashion to achieve higher total processing power. The other factor is the tremendous success of several commercial relational database systems such as Ingres, Oracle, and Sybase. And as discussed, queries submitted to these relational databases are ideally suited for parallel execution. Another advantage that comes with utilizing parallel architectures is scalability and incrementality, which, otherwise, would be difficult to achieve on mainframe computers. As a consequence, the main stream solution to VLDB nowadays is to build parallel/distributed databases.

As the demand grows for building machines powerful enough to meet the CPU and I/O requirements of large databases serving a large number of concurrent users and maintaining terabytes of data, several parallel database systems have been built for commercial or research purposes. Some of these are: Teradata, Tandem, Gamma, Bubba. Commercially successful database companies are also turning toward parallel databases. For example, the Oracle database system now has been implemented on top of a 64-node nCUBE shared-nothing system. The system demonstrates substantially better performance than traditional database built on mainframe computers, both in terms of peak performance and cost/performance ratio.

The major design issue for parallel databases is how to design query execution scheme to fully utilize parallelism inherent in these parallel systems. The objective is to implement the parallel database machine that has the ideal properties of linear speedup and linear scaleup. Below, we will present definitions of these two important criteria that are used to evaluate the performance of parallel and distributed database systems.

Given a fixed job, speedup is measured by

$$\text{Speedup} = \frac{\text{Small_system_elapsed_time}}{\text{Big_system_elapsed_time}}$$

Linear speedup is achieved if N-times larger system gives a speedup of αN for some constant α .

$$\text{Scaleup} = \frac{\text{Small_system_elapsed_time_on_small_problem}}{\text{Big_system_elapsed_time_on_big_problem}}$$

linear scaleup means that scaleup is equal to 1.

To achieve the desired properties, we must provide load balancing solutions that reduce data skew because the most skewed processor hampers the completion of the whole job. We must also avoid bottlenecks in accessing shared resources. These are difficult problems that are responsible for performance deterioration of parallel systems. Chapter 2 will discuss efficient implementation of some parallel algorithms that alleviate these problems. An effective distributed query processing strategy should be designed to minimize the amount of data transfer and/or improve response time to queries submitted by users. The effort pursuing such objectives is complicated by the issues that are inherent within a distributed computing environment such as communication delays. This is especially true in long-haul networks because these networks tend to prohibit the fast distribution of status information.

The emergence of fast communication networks (ATM, Myrinet, etc.) is what brought life into some once thought infeasible systems such as Network of Workstations.

1.2 Network of Workstations

NOW is becoming a viable platform to build parallel databases as opposed to expensive Massively Parallel Computers (MPPs). It is known that typical office computer applications do not fully utilize a workstation's processing capacity. Studies have shown that, on average, a workstation's central processing unit is idle 91% of the time [38]. The

idea of utilizing idle resources over a network has thus emerged. Parallel computing on a cluster of workstations is not a new idea. What is new is the recent technological advances and the convergence of technologies and system concepts that together make NOW a more attractive option than before. This section explains the rationale behind the renewed popularity of using NOW as a viable alternative to supercomputers in the area of parallel/distributed computing as well as the difficulties that are inherent in making such systems more efficient.

1.2.1 Historical Development

In the past, networks of workstations were a loose collection of workstations physically scattered across users' desks. The main research interest at the time was to effectively exploit the idle resources when the users were not running jobs on their desktop machines [64]. Due to the expense of finding idle cycles and migrating jobs in case that workstation's owner returned, such systems performed only batch-processing, throughput optimization for long-running computationally intensive applications. Furthermore, the shared bus media in Ethernet and high-overhead general-purpose communication protocols restricted the use of NOW to only sequential and coarse grained parallel applications. In baseline NOW system that assumes Ethernet (10 Mbps), PVM (Parallel Virtual Machine, a popular message-passing system for workstations) and a

sequential file system (2 Mbytes/second), the performance is dreadful as it takes three orders of magnitude longer than those parallel machines. A single bus-based Ethernet seems to be the bottleneck limiting the performance of NOW.

However, with the rapid advances in new high-speed general-purpose computer networking technologies (e.g., Fast Ethernet, ATM, Myrinet) and the development of more powerful and cost-effective workstations, NOW is becoming increasingly competitive compared to expensive parallel machines. In the last few years in particular, massively parallel computers (MPPs) and NOWs have become more alike. MPPs are using the same processors, memory and even operating systems found in workstations, while NOWs have started to use low latency, general-purpose LANs similar to MPP-style networks [9]. As a consequence, the performance gap between workstation clusters and supercomputers is narrowing. The workstation approach provides a relatively low-cost, low-risk entry into parallel computing arena. It incurs little additional cost to most organizations as many organizations already have extensive workstation base. No special hardware is required to use this facility as a parallel computer, and the resulting system is scalable, easily maintainable and upgradable. NOW system nowadays delivers comparable performance to supercomputer in many areas of parallel application [2]. As predicted by Anderson et al. (1995), upgrading the Ethernet to a higher bandwidth ATM network (155 Mbps) improves the interprocessor communication by an order of magnitude. Adding a parallel file system that delivers 80 percent of the aggregate

bandwidth of the workstation disks improves the overall performance yet another order of magnitude. Finally, replacing PVM with even lower overhead, lower latency communication protocol further improves performance by one order of magnitude. Overall, NOW system can compete with MPP in many application areas at a fraction of MPP's cost. In other words, by relying solely on commodity hardware and software, networked workstations can offer parallel processing at a relatively low cost.

1.2.2 Issues

On the other hand, experts have predicted that these still loosely coupled multiprocessors will, by no means, replace the more tightly coupled MPPs which have the niche of being more efficient for applications with stringent synchronization and communication requirements due to lower latency, higher bandwidth system design [1]. When compared to massively parallel supercomputers such as CM-5 from Thinking Machines, network of workstations suffers an obvious weakness: network performance. In particular, the software overhead incurred when sending a message on a typical workstation is often two orders of magnitude greater than the corresponding overhead on a parallel supercomputer. The bandwidth of a typical workstation network today (e.g., Ethernet) is again two orders of magnitude less than the bandwidth of a parallel supercomputer's interconnects [8].

Sharing CPU cycles poses many other challenging problems as well, such as computer security, network management and resource management in a distributed environment. NOW also proves harder to program than dedicated parallel machines. Sharing of CPU and the network in a multi-user environment such as NOW contributes to varying performance. Heterogeneity in processors, memory and network further contributes to the complexity and the degree of difficulties in designing an efficient distributed operating system for NOW.

1.3 Motivation

The objective of this thesis is to provide a performance evaluation of three major architectures used for parallel query processing in a NOW-based environment (centralized, distributed and hierarchical architecture). Simulation programs for each architecture will be developed. Experiments will then be conducted to evaluate the performance of three architectures for query processing on a simulated NOW platform.

The performance difference as a result of using these three different architectures was previously deduced on a qualitative basis [19]. However, there was never a performance study conducted to systematically compare and quantify the performance of the three architectures. With this thesis work, we hope to address the issue by conducting

a thorough performance evaluation of all three NOW architectures in the context of parallel query processing.

1.4 Overview of the Thesis

In this thesis, we will briefly describe the three architectures (centralized, distributed and hierarchical architectures) used for database query processing on network of workstations. The behavior of each architecture will be studied in detail by performing simulation experiments. Analysis of the simulation results is followed to evaluate the performance of each architecture.

The remainder of the thesis is organized as follows: Chapter 2 gives an overview of the related work in load sharing/balancing, networking performance of NOW, and parallel query processing platforms and strategies. Chapter 3 describes the three architectures and simulation models in detail. Experimental results are presented in Chapter 4 along with the performance evaluation. Finally, Chapter 5 is the conclusions and future directions.

Chapter 2

Background

With the availability of powerful workstations and networks with high bandwidth, low latency switches, the pressing problem that must be addressed in cluster computing is management, that is, how to effectively manage the pool of resources so that the network of workstations offers computing resources more than just a group of workstations on a fast network.

In this chapter, we will focus on three major issues in designing and implementing parallel processing on network of workstations: scheduling, network performance and parallel query processing.

2.1 *Scheduling*

Parallel computing on a network of workstations presents a variety of scheduling policy choices. When given a set of parallel jobs to execute, a scheduler must answer several questions for each processor: Should the processor work on a parallel job at all? When should the processor work on parallel jobs? And which parallel job should the processor

work on? The issues that will be addressed here are: load balancing and sharing issues, parallel processor scheduling.

2.1.1 Load Balancing and Load Sharing

It has been widely observed that usage of computing resources in a distributed environment is bursty over time and uneven among hosts. Avoiding load imbalances and exploiting powerful hosts should lead to better job response time and resource utilization. Several studies in 1980s already confirmed such intuition (e.g., [34, 40]).

Load sharing is the process of redistributing the system workload among the hosts to improve performance and accessibility to remote resources. The goal of load balancing is to assign to each processing node work proportional to its performance, thereby minimizing the execution time of the application. Although the term load balancing sometimes implicates static allocation of load to processing nodes and load sharing is regarded as more of a dynamic approach, these two terms are sometimes used interchangeably in the literature. Here we will use load sharing in its dynamic sense while load balancing strategies involve both static load balancing and dynamic load balancing. The difference is that static load balancing is done at compile time while dynamic load balancing is done at run time. The distribution of tasks is complicated if processors have differing speeds and memory resources, or different amount of external load and non-uniform iteration execution time. In general, while static load balancing avoids the run-time overhead, in a multi-user environment such as NOW with load

changing constantly on each node, a more dynamic approach is necessary. The prevailing choices include sender-initiated policy where an overloaded workstation seeks an underloaded workstation to transfer work to, and receiver-initiated policy where an underloaded workstations seeks jobs from an overloaded workstation. A hybrid policy called adaptive policy is also possible, where the system behaves as sender-initiated under light load and receiver-initiated under heavy load [20]. A major challenge to the dynamic approaches, is to minimize the additional run-time overhead.

Numerous studies on load sharing were performed in 1980s (e.g., [22, 40, 41]). The performance of a scheduling program was found to be mainly affected by four overhead sources: synchronization overhead, job assignment overhead, load imbalance, and data communication. Although it is desirable to eliminate or minimize overheads on these aspects, conflicts often arise among them. Exploiting processor affinity (i.e., certain data access dependence of a task to a specific processor) favors the allocation of tasks close to its data, which tends to increase load imbalance. Load balance favors the “fine grain” allocation of tasks in order to minimize the effects of uneven assignment. However, the “fine grain” allocation tends to increase synchronization overhead and allocation overhead. In different applications, each overhead source affects performance differently. Hence, an efficient allocation scheme should optimize its performance by adaptively trading off synchronization overhead, load imbalance and data communication overhead.

The dynamic load balancing strategies fall under different models, which include the task queue model, diffusion model and schemes based on extrapolating future load from past load information. They will be discussed in turn as follows:

- Task queue model

Most dynamic balancing algorithms proposed in the literature are based on task queue model, which can be divided into central queue based algorithm and distributed queue based algorithm. Dandamudi and Chen (1995) proposed a hybrid queue model called hierarchical queue model that combines the features of central queue and distributed queue. In the centralized queue based schemes there is a central task queue which mainly targets shared-memory systems. Once the processors have finished their assigned proportion, more work is obtained from this queue. NOW systems that adopt a global address space (e.g., Distributed Shared Memory paradigm) could potentially implement such load balancing strategies. The major advantage of a central queue is that it is more likely to evenly balance the workload. However, this approach does not facilitate the exploitation of processor affinity. In addition, since all processors remotely access the central work queue, it not only generates heavy network traffic, but also results in a longer synchronization delay due to the performance bottleneck around the central queue. The simplest approach in this model is self-scheduling [63] where each processor is allocated one iteration at a time. This approach leads to high synchronization cost, which is highly unamenable for NOW systems. To

alleviate the problem, fixed size chunking [39] and guided self-scheduling [53] could be used instead. In fixed size chunking, each processor is allocated K iterations at a time instead of one, while in guided self-scheduling, the chunk size changes at run-time.

Other schemes in this model are adaptive guided self-scheduling, trapezoidal self-scheduling, safe self-scheduling, tapering. Adaptive guided self-scheduling [26] includes a random back-off to avoid contention for the task queue and assigns iterations in an interleaved fashion to avoid imbalance. Trapezoidal self-scheduling [65] linearly decreases the number of iterations allocated to each processor. Tapering [45] uses an execution profile to select a chunk size that minimizes the load imbalance. And safe self scheduling [44] uses a static phase where each processor is allocated a chunk of iterations, and a dynamic phase where the processors are self scheduled to even out the load imbalances. To determine the applicability of these load balancing strategies to NOW systems, note the fact that these strategies are mainly designed for the systems with DSM paradigm [11], and be cautious when applying the strategies that involve high synchronization cost because NOW as a loosely coupled distributed system can not afford high synchronization cost. Finally, to exploit more processor affinity that is lacking in the above schemes, affinity scheduling [46] was proposed to take processor affinity into account while scheduling. Under affinity scheduling, all the processors are initially assigned an equal-sized chunk taking into account data reuse and locality. Once the local chunk is finished, an idle processor finds the

most overloaded processor and removes $1/P$ of iterations (where P is the number of processors) from this processor's work queue. It is shown that this scheme almost achieves the best performance in all tested cases when compared with those central queue based algorithms previously proposed. This affinity scheduling algorithm is enhanced further by adding the capability of adaptively adjusting allocation granularity according to the workload and execution speed [68]. However, this latest variation of affinity scheduling is suitable only for small system size due to the run-time overhead to collect information from different processors.

- Diffusion model

Tasks in the diffusion model are initially distributed, then tasks will migrate between adjacent nodes if an imbalance is detected between them. An example is the gradient model [43].

- Future prediction model

This model tries to predict future performance based on past load distribution information. In a global distributed scheme - Data Parallel C [50], loop iterations are mapped to virtual processors, and these virtual processors are then assigned to the physical processors based on past load behavior. The scheme is globally distributed because the processor's load is given as the average computation time per virtual processor, and the load balancing involves periodic information

exchanges. CMU also implemented a parallel program DOME for dynamic load balancing on a network of workstations. DOME adopts both global centralized scheme and local distributed scheme [3, 58]. In Phish - a research prototype implemented at MIT [8], a local distributed receiver-initiated scheme is used, where the processor requesting more works chooses a “victim” at random from which to steal more work rather than wait for work to be assigned. If the current victim cannot satisfy the requirement, another victim is selected. CHARM is another system implemented with local distributed scheme in its dynamic phase preceded by a static phase when work is assigned to the processors based on speed and load [56]. The information exchanged in all these future prediction schemes is the Forecasted Finish Time (FFT), i.e., the time for the processor to finish the remaining work. If the FFT falls below a threshold, the node sends a request to a neighbor with higher FFT for more work. If the work cannot be satisfied, another neighbor is selected. Performance evaluation comparing these schemes on a network of workstations finds that, different schemes are best for different applications under various program and system parameters, such as the number of processors, data size, iteration cost, and communication cost [69]. It was thus concluded that customized load balancing schemes become essential for good performance. Thus, Zaki et al also presented a hybrid compile time and run time model with a decision process which selects the best scheme, along with automatic transformation of sequential code to parallel code with the appropriate calls to the runtime load balancing library. In

this approach, a synchronization is triggered by the first processor that finishes its portion of the work. This processor then sends an interrupt to all the other active slaves, who then send their performance profiles to the load balancer.

2.1.2 Processor Scheduling Policies

Parallel scheduling is composed of at least two independent steps: the allocation of processes to processors (space-sharing) and the scheduling of the processes over time (time-sharing). A large number of studies have focused on the processor allocation step of parallel job scheduling. Time sharing issue is usually addressed on how to time-share between competing jobs on the same processor after jobs have been allocated to processors. Space sharing is a popular approach for optimizing the throughput of the compute-bound, production-level jobs. It does not, however, efficiently handle interactive or I/O intensive workload because the processor remains idle when a job is blocked waiting for I/O to complete. Typically, space-sharing policies are good when job parallelism does not vary much at low to medium system loads, whereas time-sharing policies perform better with heavy system loads and widely varying job parallelism. Dandamudi and Thyagaraj (1997) proposed a hybrid policy that is based on hierarchical run queue organizations that make use of both space-sharing and time-sharing techniques to achieve better performance.

When time-sharing parallel jobs, traditional MPPs use explicit coscheduling, that is, all the threads in a job are scheduled (and descheduled) simultaneously. Coscheduling or gang scheduling arises in the context of scheduling groups of processes that need to be synchronized to one another for reasons such as data exchange between tasks, or monitoring the progress of the computation [51]. This resource management method divides a large task into subtasks that are scheduled to run concurrently on a set of workstations with coordinated time-slicing between different parallel jobs. The algorithm is best suited for single program multiple data (SPMD) parallelism, where the same program runs on multiple workstations with data partitioned between the workstations. This type of parallelism generally is more flexible than multiple instruction multiple data (MIMD) parallelism where different workstations execute different tasks. The flexibility resides in the fact that there is typically more freedom in partitioning the data than partitioning the task or program. SPMD is viewed as a large grain version of single instruction multiple data (SIMD) parallelism, with each large grain processors emulating several thousand SIMD processors. In this case, partitioning the work between workstations consists of specifying the number of SIMD processors to be emulated by each workstation. This is not always a trivial task as we must take into account the amount of network traffic generated by such partitioning method. If, however, the task is compute-intensive rather than communication-intensive, then the amount of communication becomes less sensitive to the partitioning method [37]. The major advantage of coscheduling is that a parallel job runs simultaneously on workstations, allowing the job's communication requests to be serviced without a context switch.

The coscheduling idea since its introduction in 1982 has been included in many studies of multiprocessor scheduling techniques (e.g., [15, 32]). On workstation clusters, Purdue University researchers [7] presented an algorithm with $O(N^2)$ time and $O(N)$ space for choosing the optimal subset of workstations out of N workstations. In this algorithm, the process working set of a task group is not given a priori but is determined as a result of a bidding process. This algorithm is later improved to $O(N \log N)$ time and $O(N)$ space by Efe et al., 1995.

Another approach to scheduling parallel jobs on NOW is to allow the underlying OS on each workstation to schedule the parallel applications independently; we term this local scheduling. This policy, employed by parallel environments such as PVM, has the advantage of no kernel modification required. However, early studies comparing coscheduling and local scheduling have unanimously concluded that local scheduling is insufficient for fine grained parallel application [4, 15, 32]. Local scheduling has poor performance because cooperating processes are not scheduled simultaneously across the processors. Consequently, when a process attempts to communicate or synchronize with another process, the process may not be scheduled. Thus the programmer or run-time system must choose an action after initiating a communication request: spin-wait until the response is received or block and switch to another competing job. If spin-wait is chosen, CPU time is wasted while the process spins idly; if process blocks, time is wasted as well in context switching and thrashing between competing processes.

In general, the performance of local scheduled parallel application degrades dramatically when one or more other parallel applications are running on the same NOW.

Coscheduling of parallel applications guarantees an acceptable level of performance regardless of the number of competing applications [4]. Unfortunately, explicit coscheduling has a number of disadvantages that are accentuated in the NOW environment. First, it requires a centralized master to determine a fixed schedule for running parallel jobs across workstations. Ensuring that context switches occur at the same time on all processors increases the cost of each context switch. Second, explicit coscheduling does not interact well with interactive jobs or parallel jobs requiring extensive I/O. Finally, explicit coscheduling requires that communicating processes are identified a priori, and so it can not be applied to distributed client/server applications. Thus, some researchers proposed a variation of the coscheduling called implicit coscheduling [5, 6] or dynamic coscheduling [59]. Both of these approaches appear to have many of the advantages of explicit coscheduling without the disadvantages mentioned above. Unlike explicit coscheduling, both are completely distributed, requiring no global coordination. Neither requires that communication processes be statically identified. Implicit coscheduling avoids the problems of traditional coscheduling by using communication and synchronization occurring naturally within the application to coordinate scheduling across workstations. The philosophy behind implicit coscheduling is that communication and synchronization events within the parallel application provide sufficient information for coordinating the scheduling of cooperating processes. By observing the interval that processes wait for communication or synchronization events to complete, each local scheduler is able to make independent decisions that tend to schedule the processes of a parallel application in a coordinated manner across processors.

2.2 *High Performance Networking of NOW*

The emergence of new general-purpose high-speed networks is the key enabling factor that makes the use of network of workstations a possible platform for parallel processing. Although the gap between NOW and multiprocessors is certainly closing due to the availability of these high speed networks to interconnect increasingly powerful workstations, the major difference between NOW and MPP remains to be the network performance. Limited network performance could lead to excessive message delays and poor parallel application performance.

When investigating network performance, it is important to recognize that the time per communication operation breaks down into portions that involve different resources: the processor, the network interface, and the actual network. In a system like NOW where processors physically communicate by point-to-point messages, the communication cost is characterized by the following parameters [48].

- the latency, or delay, incurred when communicating a message from source processor/memory to its target processor/memory. The latency can be further broken down to the time for network interfaces to process the messages that are to be transmitted or received and actual transit time through the network. The latter includes the time to acquire transmission access to the network or switch and propagation delays. Switch delays can be attributed to signaling, routing translation

and buffering delays. Propagation delays include delays both across the switching fabric and across the links.

- the overhead, defined as the length of time a processor is engaged in the transmission or reception of each message. This includes software protocol processing, buffer management, context switching (e.g., thread or process switching and interrupt handling) and handshaking with the network interface such as data transfer using direct memory access (DMA) or programmed I/O (PIO). During this time, the processor cannot perform other operations.
- the gap, defined as the minimum time interval between consecutive message transmissions or consecutive message receptions on a module. This is the time it takes for a message to cross through the bandwidth bottleneck in the system.

2.2.1 Switch-based Networks

In recent years, there has been a technological evolution from low-bandwidth shared media local area networks, such as 10 Mbps Ethernet and FDDI (Fiber Distributed Data Interface), to switch-based local area networks, such as ATM and Myrinet. The traditional Ethernet (10 Mbps) and even fast Ethernet (100 Mbps) seem insufficient in supporting parallel processing on NOW. Studies have shown that the use of Ethernet on NOW system is the major bottleneck in performing parallel computation on the system, especially those communication-intensive tasks. Such networks are proportionally slower than parallel computer interconnects, in terms of latency and bandwidth. As a

result of sharing the same network media, the aggregate network bandwidth is approximately the same as the bandwidth of individual links. Consequently, parallel programs can place very high sustained communication loads on NOW if these traditional networks continue to be used, causing congestion and high communication delays. This phenomenon not only severely limits the number of workstations that can fruitfully participate the parallel computation but also limits the granularity of the application. While we are still using the slower Ethernet technology, today's high performance workstations have more than one hundred times the performance of their counterparts of a decade ago. At present, therefore, most research on NOW is conducted on ATM networks or Myrinet networks. Both ATM and Myrinet are switch-based networks which use point-to-point links to connect switches in an arbitrary topology that allows aggregate bandwidth much higher than the bandwidth of a single link. Because hosts are connected to the network via a dedicated link, performance does not degrade as sharply as network load increases. Throughput can be increased incrementally by adding more switches and links to match workload requirements. The switch-based network no longer has the shared media bandwidth as a bottleneck of network performance. In other words, the aggregate traffic capacity can potentially increase with the number of hosts. This will greatly alleviate the network contention problem¹. The following is a brief introduction of these two networks.

¹ This is not to say that there is no network contention problem in switch-based networks as the contention here may result from higher queueing delays in the switch and host buffers, longer time to access shared switch buffers, etc.

2.2.1.1 Myrinet

The NOW interconnect that closely resembles the MPP interconnect is the Myrinet. Today, both Berkeley NOW project and Wisconsin COW project deployed Myrinet on their NOW systems. Myrinet employs the same technology used for packet communication and switching within massively parallel processors. A Myrinet LAN connects workstations or PCs through full duplex 1.28+1.28 Gigabits per second links and low latency cut-through switches. It employs the state-of-art wormhole routing algorithm. Messages have a header flit, routing flits, a data payload, an 8 bit CRC, and a tail flit. Upon the arrival of a message, a switch uses the next routing flit to direct the message to the specified output port. Should a message block and wait for an output port, the rest of message may remain in the network, occupying switch and link resources. When the network is lightly loaded, a message can form a circuit from the source to the destination. Under heavy load of small messages, however, the network may behave like a packet switched network. Myrinet is most often used as a LAN of relatively small physical diameter tying together computers distributed across a room or a building. Myrinet link can operate on flexible, shielded, multiconductor cables up to a distance of 10 meters. Due to its high data rate, low latency cut through switching and moderate cost (\$1,600-1,800 per host), Myrinet is particularly popular for connecting clusters of computers. Myrinet is also a System Area Network (SAN) that can cluster computers within a cabinet with the same performance but at an even lower cost than a Myrinet LAN. SAN links also operate on 1.28+1.28 Gbps with very flexible, light weight,

microstrip cables up to 3 meters. Myrinet components are implemented with the same advanced technology - full custom - VLSI CMOS chips - as today's workstations. This ensures that Myrinet advances in step with the advances in the hosts, without changes to network architecture and software.

2.2.1.2 ATM

ATM LAN is another candidate. It is currently operated at a typical 150 Mbps each port (OC-3 speed), but much higher bandwidth is also available, such as OC-12 (600 Mbps/port), OC-48 (2.4 Gbps/port). Due to its nature as a switch-based network, its aggregate bandwidth grows with the system size. The basic communication unit is called cell, which has a fixed size of 53 bytes. Larger messages have to be segmented in order to be carried on those cells. The switching technology is based on virtual circuit concept, but the detailed implementation is mainly proprietary, depending on vendors. The standardized routing and signaling protocols are already underway. The two major standardizing bodies are ATM Forum and ITU-T. A performance evaluation of ATM network used on NOW is available [35].

2.2.2 Communication Protocols

The development and deployment of high performance local area networks such as ATM [23] and Myrinet [9] have the potential to dramatically improve the communication performance for network applications. These networks are capable of microsecond latencies and bandwidth of hundred of megabits per second, even gigabits per second. Their switched fabrics eliminate the contention seen on shared-bus networks such as traditional Ethernet. Unfortunately, this raw network capacity is not fully utilized with the current communication software. While higher link bandwidth and much higher aggregate bandwidth available in switched networks may benefit the performance for some applications (especially those using large messages in data transfer), many other classes of application involving request-response communication pattern and small messages may not be able to take full advantage of it due to the software overhead needed to send or receive a message [36]. A more systematic study on the effect of communication latency, overhead and bandwidth on NOW-based system reveals that applications performance shows strong sensitivity to overhead [48]. In fact, most applications demonstrate a highly linear dependence to overhead, indicating that further improvements in this aspect of communication performance will continue to improve the application performance.

There are two components of local area network overhead: software protocol overhead and network interface hardware overhead. Network interface has a hardware component (the NI) and a software component. Software components of network interface involve lower layer communication protocols processing. A study at Berkeley NOW finds that, with “hot interconnects” such as Myrinet, the performance of user-to-

user communication is primarily limited by the network interface, rather than switches and links [16]. A diverse set of hardware designs has emerged, differing on key issues such as where the NI connects to the processing node and how much processing power is embedded in the NI. However, with the NI software overhead in lower communication layers (several hundred to several thousand instructions per message), the improvement done on the hardware NI becomes obscured. Another study assessing the ATM networks used in NOW system also reached a similar conclusion [35]. The study also finds that the bottleneck of ATM network used in NOW system is at the ATM interface, rather than switch latency or link bandwidth. However, both studies state that the optimization of communication protocol, especially lower layer communication protocol should improve the performance of underlying interconnect substantially.

Therefore, there has been some work done to create “lean” communication layers. Overheads from protocol software, such as the ubiquitous TCP/IP (reliable and connection-oriented) and UDP/IP (unreliable, connectionless) protocol suites, can be further broken down into per message and per byte overheads. Examples of per-message overheads are protocol control block manipulation, software interrupt handling, context switch processing and protocol-specific processing (e.g., address resolution or TCP flow control). Examples of per-byte overheads are data checksumming and data movement (e.g., user-to-kernel or kernel-to-network interface). TCP/IP is designed to achieve good throughput in wide area networks. However, TCP/IP’s observed performance has generally not scaled up to the ability of modern network hardware, although the performance differs from implementation to implementation. While TCP/IP can achieve

good throughput on currently deployed networks, the round trip latency of commercial TCP implementations is hundreds of microseconds higher than the physical limits of these networks. Furthermore, the observed bandwidth and round-trip delay using TCP/IP on next generation network technologies such as ATM and Myrinet do not even begin to approach the raw capacity of these new networks. The overhead in TCP/IP mainly comes from its multiple layers design which separates the protocol and interface stack into multiple layers. The sockets interface is usually the topmost layer, sitting above the TCP/IP layer [55]. Implementation of simpler UDP protocol, which lacks the reliability and ordering mechanisms of TCP, performs a little better [30]. The other two popular communication layers: MPI (message passing interface) and PVM (parallel virtual machine, Sunderam 1990, 1996) also impose significant overhead [16]. Traffic pattern of local area network further exacerbates the weaknesses of these protocol suites. For example, most LAN traffic uses small packets and most application level protocols in LAN (X11, FTP, NFS etc.) operate in a request-response, or client-server, manner.

Traditionally, there are two methods to tackle the overhead problem: 1) change the application programming interface (API); 2) change the underlying network protocol, or change the implementation of the protocol. Changing the API requires the modification of the code used by applications to access communication functionality. While this approach may yield better performance for new applications, legacy applications would have to be reimplemented to have this benefit. Changing the communications protocol means modifying the packet format or processing steps during communication. This creates the incompatibility between the network that uses the new

and old protocol. Changing the implementation of protocol has the least of incompatibility problem, but the performance gain from it may be less as some fundamental flaws are embedded in the protocol design.

Recent systems, such as Active Message [28, 29, 30], Remote queue [10], and native U-Net [31], have adopted the implementation of new protocol and new API to improve network performance. Active Message provides simple communication operations with overhead limited to tens of instructions per message. Each Active Message layer is carefully optimized for a particular hardware platform, although the user communication operations are kept the same [47]. Remote Queue model was introduced as a communication model equivalent in power to Active Message in its simplest form with polling; it is thus simpler than full Active Message. Basically, it separates the queuing of messages from the invocation of handlers, which allows new optimization with this clean atomicity model. U-Net is also implemented with a light weight protocol and light weight interface. It virtualizes the network interface, allowing multiple processes to share the interface. All three models provide programming abstractions that are similar to the underlying hardware. They all realize latencies and throughput close to the physical limits of the network. However, due to the methods they adopt, none of them offers compatibility with the existing applications. There were efforts made to reimplement TCP layer with the same TCP interface on U-Net interface von Eiken et al., 1995b) to deal with the compatibility problem, but round trip latency rises considerably after adding the TCP layer. The most recent attempt was the Fast Socket developed at UC Berkeley [55]. It also uses a light weight protocol and efficient buffer management

to minimize cost. The communication protocol and API are integrated to eliminate module-crossing cost. It eliminates copies within the protocol stack by utilizing the knowledge of packet memory destinations, so called receive posting. To solve the compatibility problem, it uses the low overhead protocol for local area use, and the software transparently converts to standard TCP/IP protocols for wide area communication. In comparison, U-Net emphasizes improving the implementation of existing communications protocols whereas Fast Socket uses a new protocol just for local use. Fast Socket also has limitations. It is a user-level library, which limits its full compatibility with the Socket abstraction. First, applications have to be relinked to use Fast Socket. Secondly, it is currently single threaded. While multithreading offers one solution, it makes the library less portable and poses high synchronization cost. More seriously, Fast Socket currently cannot be shared by two processes (e.g., via fork () system call). This poses problems for Internet server daemons and for “super-server” daemons such as inetd, which depend on a fork() to process incoming requests. Research is still underway.

2.3 *Query Processing Strategies*

2.3.1 Query Processing of Parallel Databases

2.3.1.1 *Parallel Join Algorithms*

I Parallel nested-loop join

Assuming relation R join Relation S on $\{P_1, P_2, \dots P_n\}$ n processors, the algorithm is as follows:

- each node with S (inner relation) fragment broadcasts the fragment to all processors²;
- redistribute R (outer relation) among all processors;
- each processor P_i joins its local fragment R_i with the entire S relation using a uniprocessor join method (nested-loop, sort-merge, or hash join).

Uniprocessor nested-loop differs from parallel nested-loop in the way that the grain of processing is at a finer level: tuples, while the parallel version of it is at the fragment level.

II Parallel sort-merge join

Sort-merge join method consists of two steps:

- sort both relations on the join attributes;
- scan the sorted relations in parallel and merge the matching tuples.

² This step is not needed in shared-disk system.

Sorting two base relations in the first phase is easily parallelized. Parallelism in the merge phase is much harder to achieve. Two base relations are partitioned to generate a set of sorted runs. Instead of merging those sorted runs into two entirely sorted relations and merge-joining these two sorted relations, merge-join is directly applied to runs generated from sorting each partition. To increase parallelism in the second phase, either a brute-force method or a fragment-replicate [12] method could be used. Brute-force method proceeds to perform $(m \times n)$ merge joins on m sorted runs of R and n sorted runs of S concurrently on $m \times n$ processors. Fragment-replicate method first merges the smaller relation (say, R) into an entirely sorted relation and then this entirely sorted relation merge-joins with each sorted run of S concurrently on multiple processors. Although the two methods improve the parallelism in the second phase, they suffer from the increased total processing time as the merge-join becomes nested-loop in nature. The advantage of the sort-merge-based algorithm is that it produces sorted join results. Furthermore, if the base relations are already sorted on the join attributes (they are in many cases), the first phase of sort-merge join could be saved.

III Parallel hash-based join

Uniprocessor hash-based join is probably the most suitable technique for parallelization, because it partitions the two base relations into collections of disjoint subsets that can be processed independently and in parallel [66]. Here is a description of this algorithm (Assume join condition is $R.A=S.B$):

- partition R into n buckets using suitable hashing function on R.A
- partition S into buckets using the same hash function on S.B (so that if an S tuple and an R tuple do not fall in the same bucket, they could not have matching A attribute and B attribute. Only the R tuples and S tuples that fall in the same buckets are possible to have matching tuple and be in the join result);
- for each bucket i do
 - for each tuple r in R_i do
 - (each tuple still carries a marker to indicate which relation they belong to)
 - hashing on r.A and insert into a hash table (using a different hash function)
 - for each tuple s in S_i do
 - hashing on s.B and probe the hash table built for R tuples
 - if $s.B=r.A$, output tuple (r,s)

Research has indicated that hash-based algorithm outperforms the other two types of join algorithms in most cases due to the fact that the tuples belonging to the different bucket after hashing are eliminated since they do not have matching join attributes. Thus, all processors can build hash table and probe hash table independently once all the tuples belong to the buckets are stored in its local disk. As a result, this algorithm incurs least amount of interference. However, hash-based join poses its own problems. First, a suitable hash function has to be found to ensure a reasonably good partition that

guarantees the size of each bucket to be as large as possible to make full use of local memory, but not too large to fit in the memory, in which case bucket size tuning is needed for this bucket overflow problem. Rehashing of the overflowed bucket may be necessary under such circumstances.

2.3.1.2 Architectures of Parallel Database Systems

Parallel databases are typically based on three architectures: Shared-Disk, Shared-Nothing and Shared-Memory. Among the three architectures, Shared-Memory architecture is gradually becoming obsolete. Therefore, we will mainly discuss Shared-Disk and Shared-Nothing architectures in this section.

I Shared-Disk Architecture

A schematic diagram of shared-disk system architecture is shown in Figure 4. Each processor has a private memory but has direct access to all disks. The IBM Sysplex and original Digital VAXcluster typify this design [24].

Nowadays, more and more database designers are inclined to adopt shared-nothing system over shared-memory or shared-disk system. And between shared-disk and shared-memory, the former seems to be a better option. In fact, many shared-memory systems adopt shared-disk strategy at the same time in order to ameliorate interference problem which is the logical consequence of affinity scheduling which gives

Figure 4 Shared-Disk Architecture

each process an affinity to a particular processor. Research has shown that, if the disk interconnect network can scale to thousands of disks and processors, then a shared-disk design is adequate for large read-only databases or databases that do not need to perform concurrent updating tasks. Otherwise, system must be able to solve the consistency problem that comes with concurrent writing to the same physical copy by exchanging reservation messages (acquire and release lock on portions of data referenced) and exchanging large physical data pages. This creates heavy traffic on the shared interconnection network.

In shared-disk system, the implementation of parallel nested-loop join algorithm is rather straightforward. Since the inner relation is accessible by all processors, the step that involves broadcasting the inner relation to all processors is saved. Then each processor processes a portion of the outer relation and joins it with the entire inner relation. The only design consideration is to reduce the disk access contention and repeated loading of inner relation from disk. One solution is to utilize index on the join attribute if such index exists for one of the relations [42]. It is suggested that the relation with the index should be treated as outer relation, the relation without the index can be replicated in whole to each node as inner relation, then each processor simply loads the index of local outer relation fragments into memory and compare the index with join attribute of every inner relation tuple. It will retrieve the actual outer relation tuples from disk only when a matching join attribute is found. This way, disk access is reduced as index file is usually much smaller than actual data file which often can fit into memory for the purpose of comparison. As discussed earlier, sort-merge join becomes nested-

loop in nature as a consequence of trying to increase parallelism in the merge join phase. Therefore, the implementations of parallel nested-loop and parallel sort-merge are similar in many ways. The partitioning phase of hash-based join in shared-disk system is easy to implement due to the fact that all the disks are accessible by all the processors. During the partitioning phase, each processor will read pages of R and S relation from the shared-disk and hashes tuples into different partitions. The tuple that has been hashed to the same partition will be transferred to the same site (i.e., processor)³. After the partitioning phase, each processor will possess all the tuples that belong to one partition stored in its local disk. The joining phase is implemented by the following steps: each processor simultaneously builds a local hash table for one relation, preferentially using different hash function; The same function is subsequently applied to tuples from another relation and the hash result is used to probe the hash table to find the matching tuples. As indicated by the implementation of the first phase, multiple processors concurrently reading pages from the shared disk is bound to incur disk I/O contention, a problem not exist in shared-nothing systems.

Performance evaluation of these parallel join algorithms in shared-disk environment has been conducted [66]. The study finds that the implementation of the nested-loop join algorithms turns out to be the simplest and its execution time is found to be inversely proportional to the number of processors. Therefore, it works best when the number of processors is very high. Furthermore, since each tuple is compared with every tuple in another relation, the nested-loop join also supports non-equi-join. The sort-merge

³ Without shared-memory, building a global hash table is not possible. Therefore, the tuples that do not belong to the processor have to be transferred to the processor, where the bucket to which the tuple belongs

join algorithm is more complex and performs better as the relation becomes large. Given a certain configuration, the addition of new processors makes little improvement on the performance. The sort-merge join algorithm is the simplest if the operand relations have already been sorted. It also has the merit of yielding a result relation that is sorted on the join attribute. The hash join algorithm is better when the number of matching tuples in the larger relation is small. It generally performs better than the other two algorithms. To summarize, it seems that each algorithm can excel given the particular characteristics of the operand and result relations. A well designed parallel database would implement various algorithms, and chooses the best one for each join by computing their execution time.

II Shared-Nothing Architecture

A schematic diagram of shared-nothing architecture is shown in Figure 5. Each memory and disk is owned by some processor that acts as the server for that data. Mass storage in such an architecture is distributed among the processors by connecting one or more disks. The Teradata, Tandem and nCUBE machines typify this design. It is by far the best parallel architecture. Shared-nothing architecture minimizes interference by minimizing resource sharing. It also exploits commodity processors without needing powerful interconnection network. Unlike other architectures that move a large amount

is to be joined.

Figure 5 Shared-Nothing Architecture

of data through network, it moves only questions and answers through the network. Raw memory accesses and raw disk accesses are performed locally in a processor and only the filtered (locally reduced) data is passed to the client program. This allows a more scalable design by minimizing the traffic on the interconnection network. Compared to shared-disk and shared-memory systems, the major advantage of shared-nothing multiprocessor is that it can be scaled up to hundreds or possibly thousands of processors but incurs very little interference problem that severely hinders the scalability of other parallel systems. The database systems that have been constructed based on shared-nothing architecture have demonstrated near-linear speedup and scaleup on complex relational queries and on-line processing tasks. Given such proven results, database designers see little justification for the hardware and software complexity associated with adopting shared-memory and shared-disk designs.

In shared-nothing systems, the implementation of nested-loop join algorithm must involve the replication of inner relation at each node of the system as tuples from outer relation must be compared with every inner relation tuple. In local area networks such as those used in many distributed query processing studies, broadcasting facility could solve this problem quite easily and reasonably fast. Again, since parallelizing both phases of sort-merge join results in sort-merge join having nested-loop in nature, the implementation of sort-merge join algorithm shared many common features with nested-loop algorithm. A large body of evidence indicates that parallel hash-based join in shared-nothing system is the most efficient method in most cases [14, 57]. In a shared-nothing system, a processor can only access the local disk and memory. The base

relations are usually prepartitioned in some way, resulting in fragments of relations stored in different sites. Generally, there is no correlation between the sites. Therefore, data transfer is involved as processors have to exchange tuples so that each processor will have all the tuples belonging to the bucket assigned to it. During the partitioning phase, each processor reads tuples from its local fragment of the relation and applies the hash function simultaneously. Since they are not reading from a shared-disk, disk I/O contention problem does not exist in shared-nothing system. A predetermined split table is then used to route a tuple to the right processor it belongs to. Another relation is partitioned and redistributed in the same way. The joining phase is in fact the same as uniprocessor processing at each site and similar to what happens in shared-disk implementation.

Performance evaluation of these join algorithms on shared-nothing systems shows that hash-based algorithm outperforms sort-merge algorithms in almost all cases [54, 57]. Among the hash-based algorithms, the hybrid hash join is the best. However, the performance of hash-based algorithms degrades when the value of the join attributes of the inner relation are highly skewed and the available memory relative to the size of the smaller relation is limited. In such cases, a non-hash-based algorithm such as sort-merge should be chosen. Nested-loop algorithm has the advantage that it is not affected by data skew. Sort-merge join has the merit of producing the resulting relation in sorted order.

2.3.2 Query Processing on NOW

As far as applicable strategies from parallel databases are concerned, shared-disk design is considered an adequate option to be applied to NOW-based systems when we are dealing with read-only databases or databases that do not need to perform concurrent updating (Figure 2 and Figure 4). Since the purpose of using NOW to process query does not involve performing concurrent updating of the central database, shared-disk implementation is certainly a good candidate. Especially, minimum modification of shared-disk design is needed due to the strong resemblance of shared-disk architecture to NOW architecture. On the other hand, shared-nothing system appears to have a topology different from that of NOW-based system. But if we look further into the picture, the NOW system can be treated as a shared-nothing system once the workstations finish downloading all the referenced relations from the centralized database. In other words, the implementation of parallel join algorithm for shared-nothing architecture should still be a valid option after the workstations received the data from the central database.

Caution should be exercised when applying interoperator parallel strategies to NOW-based systems. One example is the utilization of pipelining technique that has been shown to greatly improve the performance of parallel database systems. In parallel database systems, pipelining can be realized at tuple level. The result of one operation can be streamed into another operation on a single tuple basis. This could be done in a parallel system because of its high bandwidth bus that makes communication cost negligible. In a NOW-based system, however, pipelining can only be applied at a much coarser grain than tuple because the interconnection network that connects the workstations is a general-purpose communications network (LAN in most cases) which

has higher communication cost. Finally, in choosing proper join sequence tree, wide bushy tree performs better than most other query execution trees because it exploits the interoperator parallelism to a fuller extent [13, 19].

Significant differences, however, exist between a NOW-based system and a parallel database system. These differences are summarized as follows:

- The database itself is not partitioned across different processing nodes in NOW environment. This implicates additional overhead as data have to be moved to each workstation from the database.
- NOW environment is highly heterogeneous as each workstation may have different CPU, memory, and disks.
- Most parallel database environment uses a central scheduler and load balancer. As a result, load on each processing node tends to be fairly stable. This is often not the case on NOW.
- Parallel database systems have a dedicated, special purpose fast communication network. The network in NOW environment involves general-purpose LANs, which usually are slower and involve rather high communication overhead.

Consequently, the major performance issues involved in NOW include load balancing, fault tolerance, and communication latency [67]. It is generally recognized that the communication latency is the most important issue as NOW-based systems heavily rely on message passing for communication.

Chapter 3

Centralized, Distributed and Hierarchical Architectures

Network of Workstations provides a feasible platform for parallel query processing (Figure 6). Based on the body of literature presented in Chapter 2, we summarized three architectures that will be undergoing our performance study, namely, centralized, distributed and hierarchical. The simulation models that are detailed in this chapter adopt various parallel scheduling policies and load balancing strategies. Performance evaluation of these three architectures will be conducted based on the simulation results.

There are, however, important assumptions made in the study based on the nature of query processing on a NOW platform. In a NOW-based system, the database management system (DBMS) would be centralized since every workstation accesses data in read-only mode. Updates requested by any workstation go through the central database node. In other words, the database node manages the database while other workstations access data through this database node. Each query submitted to the workstations indicates the relations involved and all data transfer is to be done through the centralized database node. Task allocation, however, does not necessarily involve the database node depending on the specific architecture under investigation. Once a workstation is selected for certain tasks and receives the necessary data from the

Figure 6 Components of Query Processing on Network of Workstations (NOW)

database, it processes the job independently with its own local resources, i.e., CPU, memory and disk. This poses a significant differentiating factor from distributed database systems where the database itself is distributed among the workstations. To summarize, query processing using NOW involves a centrally controlled database and a group of workstations participating query processing by sending messages across interconnection network. Data security and integrity can be maintained in a way similar to traditional databases, but idle cycles in workstations can be utilized to do parallel query processing.

3.1 Centralized Architecture

3.1.1 Overview

In centralized architecture (Figure 7), one of the nodes maintains the only task run queue in the system and pre-optimized queries are submitted from various workstations to this node. This architecture operates in a master-slave mode. One workstation acts as a master (could be the database node) and the rest of the workstations work as slaves. Figure 7 shows the major components of centralized architecture. Queries are submitted at various workstations and immediately forwarded to the master node for interpretation and processing. The database node is responsible for interpreting all queries received by the system and generating optimized query execution plans. The master node, by taking

Figure 7 Centralized Task Queue Organization

the system information into account, allocates slave workstations for certain query. System information includes query load and non-query load (background load).

In this study, the processor allocation decision is made dynamically using receiver-initiated policy, which means underloaded nodes ask the master node for work. This dynamic policy is adopted due to the heterogeneous nature of NOW systems. Consequently, the processing power of faster processor can be fully utilized as they tend to finish their assigned work faster and request jobs more frequently. In the mean time, the slower processor will not be overallocated since they complete work slower and tend to request work less frequently. The work at the master node is designed to be fragments of operation enqueued into a common master job queue so that fragments of the same operation can be distributed to different slave nodes for processing, thereby realizing intra-operation parallelism. Inter-operation and inter-query parallelism are also possible since fragments from different operation and different queries are likely to be distributed to different nodes and processed in parallel. Each slave node keeps a copy of the relations from the previously processed fragment so that, in case subsequent fragment from the same operation is allocated to the node, inner relation does not have to be fetched over the network again. This design minimizes the communication cost.

Workload in the centralized architecture tends to be well distributed among the available processors in the system. As a result, it provides good performance due to its load-sharing characteristics. As the number of processors in the system increases, however, the central scheduler becomes the bottleneck. The fault tolerance of centralized architecture is also poor since the central node may well become the single point of

failure in the system. Therefore, the centralized architecture is generally recognized as suitable for small-sized systems.

3.1.2 Simulation Model

Figure 8 shows the data flow across various components of centralized architecture in our simulation.

Master node maintains a central scheduler and two master job queues. One is `master_entry_q` which contains all pre-optimized queries forwarded from each workstation. Once a query arrives at `master_entry_q`, the eligible joins within the query are immediately fragmented into fixed-sized chunks and promoted into `master_rdy_q` which contains all the chunks ready to be dispatched to slave nodes for processing. Fragmentation is done by chopping the outer relation into fixed size chunks while keeping the inner relation (usually the smaller one) intact. A join is deemed as eligible when it has no daughter joins (left join pointer and right join pointer both are NULL) or when its left and right daughter joins are completed. Take the example of join tree in Figure 9, join A and join B are eligible because they do not have daughter joins to be processed while join C remains ineligible until both join A and join B are finished. Since receiver-initiated scheme is employed in the simulation, central scheduler awaits the work request messages sent from the slave nodes. Slave nodes send work requests to

Figure 8 Data flow in centralized architecture

Join C (parent join)

Figure 9 Balanced query format

central master node only when they find themselves underloaded. Besides having slave nodes checking their load at system initial start-up, slave nodes check their load status whenever they complete a chunk in the corresponding local slave_q. The request messages will contain the information as to which slave node, how much load is on the slave node. In response to the request message, the central scheduler will dispatch the chunk at the head of master_rdy_q to the slave node requesting the work. In case of empty master_rdy_q, a slave_waiting flag is raised, so that central scheduler will

broadcast a message asking all underloaded slave nodes to send requests to master node once new chunks enter the empty master_rdy_q. Master_entry_q maintains the join tree for each incoming query up to the point when the entire query is completed. Each join also maintains a num_chunk_left counter that decrements when master node receives the notification from the slave that a chunk belonging to the join is completed. A join is considered as completed when the master node receives the “complete chunk” notification for all its chunks and num_chunk_left counter has reached zero. At this point, the join’s parent, if exists, checks its eligibility and initiates the fragmentation and chunk promotion to the master_rdy_q if the parent join finds itself becoming eligible. When the last join of the query is completed, the entire query is removed from master_entry_q, and query results are sent to the node that generates the query. Figure 10 shows the control flow process in master node.

At slave node resides the local chunk queue and local CPU, memory and disk. When a chunk is dispatched to a slave, it is first enqueued into a local slave_q waiting for its turn to reach the head of the queue. When it does, it is assessed with job processing time which includes the relation transfer time from the database node, disk I/O time as well as the actual CPU time spent on tuple comparison and actual join operation. When the chunk is completed, it checks its load status and sends a request message to master node if the node becomes underloaded. Meanwhile, a chunk complete notification is also sent to notify the master node. Figure 11 shows the control flow in slave node.

Figure 10 Control flow diagram of master process in centralized architecture

Figure 11 Control flow diagram of the i th slave process in centralized architecture

3.2 *Distributed Architecture*

3.2.1 Overview

In distributed architecture, each workstation has its own scheduler which makes scheduling decisions for the jobs submitted to the workstation (Figure 12). Due to the dynamic arrival of the jobs and varying job complexity, some nodes may be over-loaded than others. The sender-initiated policy is employed in this study to transfer work to other under-loaded workstations once the local processor becomes overloaded, although both receiver-initiated policy and adaptive policy are applicable in this architecture.

Figure 12 shows the components of distributed architecture. All query interpretation and optimization, processor allocation and scheduling are done locally at each workstation. Therefore, the bottleneck in centralized architecture is eliminated.

Load sharing in distributed systems has been studied extensively (e.g., [22, 33, 40, 41]). Whereas the strategies used for those applications cannot be directly applied to query processing, we have adapted these ideas for our study. In this architecture, each node receives and preprocesses queries locally. CPU queue length is used as an indicator of load status at each node. Thresholds can be defined to classify nodes into overloaded or underloaded. Load sharing is achieved by simply transferring work from overloaded node to underloaded node. If sender-initiated policy is used, the sender node (overloaded node) probes the rest of the nodes in an attempt to find an underloaded node which can

Figure 12 Distributed Run Queue Organization

serve as the receiver for the work. Receiver-initiated scheme is similar to sender-initiated scheme in concept except that load distribution is initiated by any underloaded node. The polling is initiated by the underloaded node in an attempt to locate an overloaded node which has jobs to transfer. In many distributed systems, sender-initiated policy has been shown to perform better at low to medium system load while receiver-initiated policy tends to perform well under heavy system load (e.g., [41]). This is the direct result of communication incurred with the two policies at different system load. The granularity of the work also impacts system performance because each job in overloaded node needs to initiate a cycle of probing to find the appropriate host for the job. Hence, the smaller the granularity of the jobs, the larger the communication cost and load distribution overhead. However, if the granularity is made large, slower processing node can be allocated a big job, load imbalance thus is more likely to occur. Besides, since the load information is distributed and gathered independently by each scheduler polling all other workstations, the load distribution decision made by each scheduler may not be up-to-date. When the transferred work arrives at the destination node, the node must accept and process the job even if the state of the node at the instance has changed to overloaded from the underloaded status at the time of the probing. Distributed architecture, therefore, is more likely to suffer from load imbalance. Additional communication overhead resulted from constant polling from all the schedulers is also a significant factor in distributed architecture. However, unlike centralized architecture, distributed architecture offers better fault tolerance and the system is more scalable.

3.2.2 Simulation Model

As opposed to centralized model, all nodes in distributed model implement a local scheduler, a master process as well as a slave process (Figure 13). A master process consists of a `master_entry_q` containing the queries submitted to this node, a `master_rdy_q` containing the chunks resulting from the fragmentation of eligible joins promoted from the queries in the entry queue. A slave process is composed of a local `slave_q` as well as its local CPU, memory and disk. The master and slave process in distributed architecture (Figure 14 and Figure 15) bear strong resemblance to the corresponding processes described in the centralized model. The major difference is that the scheduling is now done in a distributed fashion, i.e., each node makes the job scheduling decision independent of each other. This is the result of each and every node having their local scheduler, local master process and local slave process all residing on the same node, instead of the master node running master queue and central scheduler and slave nodes running only slave processes. At the implementation level, another difference is that a sender-initiated load balancing scheme is adopted rather than the receiver-initiated scheme in centralized model. Sender-initiated scheme calls for a polling mechanism. The initial implementation of polling scheme was to collect responses from all slave nodes and then pick the node that reports the lightest load to dispatch the work to. To save the communication cost associated with collecting all the

Figure 13 Data flow in distributed architecture

Figure 14 Control flow diagram of the *i*th master process in distributed architecture

Figure 15 Control flow diagram of the i th slave process in distributed architecture

responses, the original scheme was modified to collecting the responses until an underloaded node is found. The sequence in which the responses from each node come back to the sender node is random. The modified scheme reduces communication cost significantly, but is more likely to result in load imbalance than the initial polling mechanism.

In case an underloaded node is not found after polling all the nodes, a situation that is likely to occur with heavy system load, a timer is built in to poll again after certain timeout interval. The configuration of timeout interval would affect the system performance.

3.3 *Hierarchical Architecture*

3.3.1 Overview

The hierarchical architecture is based on the hierarchical run queue and hierarchical scheduling policy proposed by Dandamudi (1995, 1997). It is intended to provide a compromise or hybrid solution as opposed to centralized and distributed architectures. Figure 16 shows a set of queues organized as a tree structure with all the processor nodes as leaf nodes. When a leaf node is seeking a job, it first sends a request to its parent queue which also resides on one of the nodes. If the parent queue is empty, then the parent queue sends a further request to its parent one level further up. This process is repeated until it finds the work in a parent queue certain levels up or until it reaches the

Figure 16 Hierarchical Run Queue Organization

root queue. In order to reduce access contention at higher levels, the architecture is designed to transfer a set of work to one level down the tree so that not all the work is concentrated at higher level queues (Figure 17). This is done by transferring a set of work units one level down the tree. The size of the set decreases progressively as one goes down the tree taking into account the availability of the increasing number of queues per level.

Every time the work moves one level down the hierarchy, always $1/B$ of work units gets transferred. This way ensures that requests from empty leaf nodes are more likely to find jobs at lower level parent queues. As a system of N nodes, the work can be distributed to a maximum of $(N-1)$ nodes. Apparently, this architecture has less contention problem than centralized architecture. As described, however, this architecture incurs heavier communication overhead than centralized architecture as leaf nodes may need to navigate multiple levels of the tree to find work, although the cost does not grow with system size by $O(N^2)$ as distributed architecture does. The drawback of this architecture arises from its design intention to pull jobs down the tree to avoid high-level queue access contention as it would in fact restrict the pool of resources (the number of processors) available to process the job.

3.3.2 Simulation Model

The master queues and slave queues in the hierarchical model are arranged as a balanced tree with a branch factor of 2 (Figure 18). Each node will have a slave queue shown as

Figure 17 Job transfer process in hierarchical run queue organization

Figure 18 Hierarchical task queues for query processing with processing node 0, ..., 7

one of the leaf nodes in the tree. All but one node will have one master queue as well shown as either root node or one of the intermediate nodes in the tree. In this simulation, more than one master queues on one node is avoided. This prevents certain nodes from becoming access bottleneck if they contain more than one master queues. Local slave queues and remote master queues maintain queries, operations of queries, or partial operations⁴. The query workload generated at each workstation is first forwarded to the root master queue of the tree. Hierarchical architecture is a self-scheduling scheme, in which bottom nodes i.e., local queues send requests up the tree to seek jobs if processor nodes find themselves underloaded. The hierarchical policy operates in three modes: query transfer mode, operation transfer mode, and partial operation transfer mode (Figure 19). To further clarify the mechanism, let us assume a 64-node system with only 4 queries at the root queue (Q1, Q2, Q3, Q4). An idle node (assume node 0) sends a request up the tree until it reaches the root where it finds the work (queries)⁵. The query transfer mode is applied as long as the number of queries is more than the branching factor B. In this mode, the work units transferred between successive levels of the tree are the entire queries. In the example, Q1, Q2 are transferred down the tree while Q3, Q4 are left at the root. Subsequently, Q1 is moved further down one level. By this move, Q1 and Q2 are effectively allocated to the left half of the tree. When the number of queries⁶ is less than the branch factor⁷, query transfer mode is transitioned into operation transfer mode where a query is split into operations and 1/B of eligible operations are passed

⁴ The operation in this simulation refers to join operation.

⁵ All the queues along the path are empty.

⁶ Not partially processed queries.

⁷ A branch factor of two is assumed in this study.

Figure 19 Query transfer mode, operation transfer mode and partial operation transfer mode in a hierarchical system of 64 nodes

down to immediate descendant job queue who requested the job, and the descendant job queue passes jobs further down to its descendant. Since QI at this point is less than the branching factor, the operation transfer mode takes effect. In this mode, operations become the transferred work units. In a similar fashion, operation transfer mode is switched to partial operation transfer mode when the number of operations in the parent queue is less than the branch factor B , in which fragments of operation become the transferred work units (partial operation transfer mode). In this case, operation is fragmented into B partial operations and one of them is passed down. The fragmentation process is performed by keeping the smaller inner relation intact and chopping the larger relation into B parts so that each partial join is a join of the smaller relation and one piece of the larger relation. This pass-down process (Figure 19) is repeated until the local slave queues at the leaf nodes are encountered. When the number of jobs is greater than or equal to the branch factor, no mode transition occurs. The parent queue simply passes $1/B$ of its jobs down to its descendant queue. A minimum chunk size is defined in partial operation transfer mode to terminate the fragmentation process when the fragment has reached certain limit. Otherwise, this fragmentation process could be repeated infinitely. Figure 20 and Figure 21 depict the control flow of master and slave process. And as described, the job transfer process in hierarchical architecture realizes inter-query parallelism, inter-operation parallelism as well as intra-operation parallelism.

Figure 20 Control flow diagram of master process in hierarchical architecture

Figure 21 Control flow diagram of the i th slave process in hierarchical architecture

3.4 Network Model

To simulate the network contention, a global marker (`last_network_usage_time`) is introduced to keep track of the time at which the last communication event finishes using the network. Any event, if the onset of its communication process is before the recorded `last_network_usage_time`, will be delayed until after the recorded marker time. To implement the scheme, we introduced two event lists. One of them (`event_list`) stores the events that **WILL** occur in a chronological order based on the event finish time, the other (`event_list_tentative`) stores the tentatively scheduled events based on their communication start time. All events generated in the simulation are first enqueued into `event_list_tentative` based on their communication onset time. They will be processed in turn as simulation clock goes on. The events not requiring network communication are immediately removed from the tentative list and placed into the `event_list` in which all the events are supposed to occur without other communication events blocking them. The events that has onset of communication occurring before `last_network_usage_time` will have their communication onset time delayed until `last_network_usage_time` and be reinserted into `event_list_tentative` based on their new communication onset time. The events that have their communication onset time after `last_network_usage_time` will have the network resource available for them to use, therefore, they can proceed to be enqueued into the `event_list` based on their event finish time. In the meanwhile, network

acknowledges the network usage due to this event by updating `last_network_usage_time` to the communication finish time of this event.

Chapter 4

Performance Evaluation

In this chapter, we study the performance of centralized, distributed and hierarchical architecture for query processing on NOW platform.

4.1 Selection of Parameters

Important parameters involved in the performance evaluation are: fragmentation size (chunk size), mean query arrival rate, number of processing nodes and network speed. Queries that are submitted to the system are in a balanced query format as shown in Figure 9. Base relation size (R1, R2, R3, R4) is randomly generated using random generator function `erand48()`. The size of base relations ranges from 32,000 bytes to 320,000 bytes. This particular range is chosen because it represents an average database relation size sufficiently to reflect any trend in reality and computation time required to process such relations in our simulation is reasonable. The arrival of the queries follows a Poisson distribution with a mean arrival rate as a user-defined input parameter. Most of the experiments are conducted using over 90 batches with each batch consisting of

100-120 queries. Simulation is terminated once the total number of queries processed reaches the number as determined by the product of `num_of_batches` and queries processed per batch. Based on these batch results, 95% confidence interval is computed as an indicator of variability of the results. Simulation results are obtained using 4 nodes, 16 nodes and 64 nodes system with various network speed ranging from 10Mbps, 100Mbps and 1000Mbps.

4.2 System Scaleup

Parallel system is evaluated by two key properties (i) linear scaleup and (ii) linear speedup. Scaleup measures the ability of the growing system handling growing problem. Linear Scaleup is defined as the ability of N _times larger system to perform an N _times larger job in the same elapsed time as the original system.

Our experiment measures the scaleup under two different system load conditions characterized by two different mean query arrival rates per node. The results in Figure 22 show that the scaleup property of all three NOW architectures is excellent as a 16-node system is able to complete 4 times larger query load in the same or even less time than 4-node system with the original load. The same holds for 64-node versus 16-node system. Figure 23 shows the mean response time scaleup under a heavier system load condition. The data indicate that the additional performance is gained by employing larger system

Figure 22 System Scaleup under light system load

*centralized architecture, 4, 16, and 64 nodes, heavy system load, 1000Mbps

*distributed architecture, 4, 16, and 64 nodes, heavy system load, 1000Mbps

Figure 23 System Scaleup under heavy system load

to do parallel processing. Not only is the 16-node system able to handle 4 times larger job in the same elapsed time as 4-node system, in some cases (lighter load in particular). 16-node system completes the 4 times larger job within a shorter period than that of 4 nodes handling the original job size. From the scaleup results, the ratio of small system to large system (4 nodes, 16 nodes and 64 nodes) based on the scaleup equation is in the range of 0.70 to 2.75. The system demonstrates better than linear scaleup performance in average. The results indicate that using more processing nodes is beneficial as more parallelism can be exploited.

4.3 System Speedup

Speedup is defined as the ratio of small system response time to large system response time on the same job size. Linear speedup means that N times larger system should process the same job at a proportionally faster rate (e.g., 16 nodes vs 4 nodes). We have conducted the simulation comparing the response time of 4 nodes, 16 nodes, and 64 nodes system with the same system load, that is, the same query arrival rate per system instead of per node in our scaleup study. Simulation results indicate that, although larger system solves the same problem in significantly less time, the performance gain is not proportional to the scale of the system. In other words, 4 times and 16 times larger systems are not able to complete the same job proportionally faster than the smaller system. The plot shows that the speedup for 4 times larger system ranges from 1.25 to

2.25 and the speedup for 16 times larger system ranges from 1.75 to 2.65. Therefore, we conclude that, while speedup is gained to a certain degree by employing larger systems, the ideal linear speedup is not achieved by using more processing nodes with centralized, distributed and hierarchical NOW architectures. Figure 24 and Figure 25 show the mean response time of three architectures for a fixed query load. Figure 26 and Figure 27 compute the speedup for 4x and 16x systems in response to the same system load.

4.4 Performance Sensitivity to Fragmentation Sizes

Generally speaking, larger fragmentation size results in fewer partial operations that can be processed in parallel among different processing nodes. Thus, less intra-operation parallelism can be exploited to improve the response time. There is, however, a tradeoff between the parallelism and communication cost, both of which impact the system performance. Increasing the number of partial operations that are sent to different processing nodes incurs more communication cost which constitutes a significant part of total response time.

From Figure 28, Figure 29, Figure 30 and Figure 31, it can be observed that mean response time is sensitive to fragmentation size (chunk size). For centralized and distributed architectures, chunk size is defined as a fixed fragmentation size in tuples or

Figure 24 System response time with fixed system workload (I. low arrival rate)

Figure 25 System response time with fixed system load (II. high arrival rate)

Figure 26 Speedup for 4x system (16 nodes vs 4 nodes, centralized, distributed and hierarchical)

Figure 27 Speedup for 16x system (64 nodes vs 4 nodes centralized and distributed⁸)

⁸ Hierarchical data not available due to the limitation of computing resources at the time the experiment was performed.

bytes when the outer relation is chopped into smaller fixed pieces. For hierarchical architecture, chunk size varies depending on different stages of pass-down process as discussed in the simulation model. However, there is a minimum number of tuple comparisons per chunk defined to terminate the further split of the chunks that are already small enough. The chunk size referred in the graph for hierarchical architecture (Figure 31) is this minimum number of tuple comparisons that each chunk entails. Logarithmic scale is used to show the degree of sensitivity among different regions of chunk size. The plot indicates that the response time gradually decreases as the chunk size decreases until it reaches the lowest point at the chunk size of around 32,000 bytes (1000 tuples) or 5,000,000 tuple comparisons in the case of hierarchical architecture. The response time then sharply increases with the further reduction of fragmentation size indicating that communication cost becomes the dominating factor of system performance. Overhead and communication cost introduced by finer grain explain the result well.

The second observation is that the response time reaches a plateau once the fragmentation size reaches the maximum relation size, which is 320,000 bytes in our simulation. It reflects the fact that, as fragmentation size reaches the maximum relation size, the outer relation no longer needs to be fragmented. As a result, the entire relation in the join operation is transferred to the processing nodes irrespective of further increase of fragmentation size. Thus, intra-operation parallelism and communication cost remains static once the fragmentation size is becoming larger than maximum

Figure 28 Response time is sensitive to the fragmentation size (I. low arrival rate)

Figure 29 Response time is sensitive to the fragmentation size (II. high arrival rate)

Figure 30 Response time is sensitive to the fragmentation size (III. 10Mbps)

Figure 31 Response time is sensitive to the fragmentation size in hierarchical architecture (a. light load; b. heavy load)

relation size. Consequently, mean response time no longer reacts to the change of fragmentation size.

Note that distributed architecture seems to exhibit higher sensitivity to fragmentation size as communication cost becomes dominant. The system performance degrades dramatically once fragmentation size drops below 32,000 bytes in distributed architecture while centralized architecture in the same region remains largely constant before it drops below 16,000 bytes. The reason for the higher sensitivity exhibited by distributed architecture is the heavier communication overhead that is inherent within the architecture itself than centralized architecture. The fact that each node in distributed architecture makes independent scheduling decisions by polling all nodes within the system constantly makes this architecture more prone to any factor that might introduce additional communication overhead, in this case, finer grain data.

4.5 Performance as a Function of Query Arrival Rate

Simulation has been performed to evaluate the performance of centralized, distributed and hierarchical architectures under different system loads, i.e., varying query arrival rates. A 16-node NOW system is chosen to implement the three architectures to process queries submitted to each node. When query arrival rate is low, a given job is likely to be distributed to more processing nodes resulting in lower average response time. When

query arrival rate exceeds what system can handle, system saturates⁹ in which state mean response time escalates with time. Typically that is when the rate at which the amount of work is submitted to the system is greater than the rate at which the amount of work can be processed by the system.

In this experiment, we chose three network speeds (10Mbps, 100Mbps, 1000Mbps) and two fragmentation sizes (4,800 bytes and 32,000 bytes in centralized and hierarchical architecture; 250,000 and 5,000,000 tuple comparisons in hierarchical architecture). Figure 32, Figure 33 and Figure 34 show the mean response time of three architectures varies with query arrival rates in 1000 Mbps, 100 Mbps and 10 Mbps networks, respectively. A general observation can be made regardless of the network speed, that is, each architecture can outperform the other architectures depending on what fragmentation size was used. When all three architectures adopt the ideal fragmentation size (i.e., 32,000 bytes for centralized and distributed architectures, 5,000,000 tuple comparisons per chunk for hierarchical architecture), hierarchical architecture consistently outperforms the other two architectures under most system load conditions except in a few light load cases, whereas the performance of centralized and distributed architectures is almost equal (refer to plot C in Figure 32, Figure 33 and Figure 34). On the other hand, centralized and distributed systems could deliver substantially better performance than the hierarchical architecture when the ideal fragmentation size is not used in hierarchical system (refer to plot B in Figure 32, Figure 33 and Figure 34). According to the results in the previous experiments on the performance sensitivity to

⁹ Network utilization reaches saturation as the system saturates with 10Mbps network speed. On the other hand, server side saturation is indicated with higher bandwidth networks (e.g., 100Mbps).

fragmentation sizes, 5,000,000 tuple comparisons per chunk is close to the ideal size in hierarchical systems while 250,000 tuple comparisons per chunk used in plot Bs is less than ideal. Even with non-ideal chunk sizes used for all three architectures, performance varies widely depending on the specific size chosen for each architecture. For instance, plot As in Figure 32, Figure 33 and Figure 34 show that hierarchical architecture with non-ideal minimum fragmentation size (250,000 tuple comparisons) could still perform better than the centralized and distributed architectures with non-ideal 4.800 bytes fragmentation size.

A simplified analysis is made in an attempt to explain the disadvantages and advantages associated with the architectures that would result in the data we have obtained in this experiment. The advantage of hierarchical architecture derives from its design that avoided the pitfalls of the other two architectures. The inherent self-partitioning nature of hierarchical architecture overcomes the bottleneck problem for the global queue as perceived in centralized architecture. The superior performance of hierarchical architecture over distributed architecture is due to the hierarchical scheduling that eliminates the need to communicate to all the nodes in the entire system but to only a subset of nodes in the subtree that it belongs to. Communication cost is thus less in hierarchical systems than in distributed systems.

However, it is particularly worth emphasizing that the performance varies with fragmentation size widely. With suitable fragmentation size, centralized and distributed

Figure 32 Performance comparison of centralized, distributed and hierarchical architectures (1000Mbps)

Figure 33 Performance comparison of centralized, distributed and hierarchical architectures (100 Mbps)

Figure 34 Performance comparison of centralized, distributed and hierarchical architectures (10 Mbps)

architecture could easily outperform the hierarchical architecture that has adopted unsuitable fragmentation size as demonstrated in our experiment.

4.6 Effect of Background Load on Performance

We have assumed that there is no background load so far. However, the NOW systems are intended to be general purpose unlike those dedicated parallel systems. Thus, the effect of background load on the system performance is examined in this experiment.

Figure 35 is the result based on the simulation performed in a 100Mbps network. Background load is turned on periodically during which 50% of CPU processing power is assigned to the background load. The impact of background load on centralized, distributed and hierarchical NOW systems is individually examined to determine which architecture better handles the background load, i.e., suffers less performance deterioration with periodic background load in the system.

The hierarchical architecture exhibits slightly less sensitivity to the background load. It is not a surprising outcome considering the property of the architecture. In a centralized or distributed architecture, the jobs that are enqueued in the same master queue are dispatched to processing nodes in a strictly sequential order. This implies that a job at the end of the queue has absolutely no access to computing resources until all the jobs ahead of it are dispatched. In hierarchical architecture, the approach is entirely different. When a job request from left-half of the subtree arrives at the parent queue, it takes the first half of the jobs down for the processing nodes in the left subtree but leaves

the second half of the jobs which are originally enqueued behind the first half in the parent queue. Any idle node in right subtree can potentially take on those jobs. As a result, the jobs that are enqueued behind have equal access to available computing resources as the jobs ahead of them in the queue. The same logic applies to the background load situation. When it comes to background load, the impact of jobs ahead of the queue taking longer time to complete due to the onset of background load would have less impact to the jobs waiting behind in the queue in the hierarchical architecture than in the centralized and distributed architecture. Performance deterioration caused by background load is less in the hierarchical system than in the other two systems.

Figure 35 The impact of periodic background load on response time

Chapter 5

Conclusions and Future Work

5.1 *Summary*

Advancement of high-speed, general-purpose networking technology and improvement in microprocessor performance/cost ratio make network of workstations (NOW) an appealing alternative for parallel computing.

In the introduction, we briefly reviewed the progress of relational databases along with its basic operations. The parallelism inherent in the relational database model was discussed. We also introduced basic types of parallelism as well as the criteria adopted to evaluate the performance of parallel systems. The historical development and the rationale behind the NOW systems followed.

In Chapter 2, a summary of background literature was presented. Several parallel database systems with different partitioning techniques, parallel processing strategies, and load sharing and scheduling policies were discussed. The up-to-date progress on networking performance was reviewed. Finally, this chapter highlighted the differences between parallel database systems and NOW-based database query processing systems and discussed the issues that remain to be studied for NOW-based systems.

Chapter 3 reviewed three predominant architectures that are proposed for database query processing in a NOW environment, namely centralized, distributed and hierarchical architecture. The disadvantages and advantages associated with each architecture were briefly discussed. The simulation models used in this performance study to quantitatively examine the three architectures were described in detail. The programs simulating the behavior of the three architectures were written in C++.

Chapter 4 presented the results obtained from the simulation experiments. The results of this study confirmed that the three architectures provide feasible platforms for using NOW to process database query. All three architectures exhibit excellent scaleup (≥ 1) and significant speedups although linear speedup is not achieved with any of these architectures as with any parallel database machines. The performance of all three architectures shows strong sensitivity to fragmentation size. An ideal fragmentation size represents the best tradeoff achieved between load sharing characteristics and communication cost determined by the granularity of the data. It was observed that query response time increases as query arrival rate rises up to the point the system saturates. Based on the results from 10Mbps, 100Mbps, 1000Mbps networks, we concluded that hierarchical architecture outperforms the centralized and distributed counterparts when all three architectures adopt their ideal chunk size. A simplified analysis is provided to explain the result. It is to be stressed, however, centralized and distributed architectures with more suitable fragmentation size could deliver superior performance when compared with the hierarchical architecture experiencing performance degradation with non-ideal minimum chunk size. Hierarchical architecture is also found to suffer the least impact of

background load among the three architectures. Simulation data also support the notion that network performance is the bottleneck in 10 Mbps LAN.

5.2 *Contribution of the Thesis*

The main objective of this thesis was to evaluate and quantify the performance difference of centralized, distributed and hierarchical architecture in database query processing on NOW platform. Several previous studies have been conducted to evaluate the performance through the implementation of the architectures in 10Mbps Ethernet LAN. While these studies provided some valuable information, due to the limitation of implementation methodology, many parameters are not easily adjustable to allow for a thorough investigation of the systems (e.g., [60]). This thesis work, through the implementation of three simulation programs to simulate all three architectures, provides the first systematic and quantitative study of the performance of centralized, distributed and hierarchical architecture in the context of database query processing.

5.3 *Future Work*

Due to the large number of parameters involved, the effect of only selected parameters is examined in the thesis. This work, however, can be easily expanded to explore other aspects of centralized, distributed and hierarchical architectures. There are several directions future studies can take based on the results obtained in this study.

One interesting scenario would be to investigate ways to further improve the performance of hierarchical architecture. One scenario is to use more bushy hierarchy by changing the branch factor of the system from 2 to 4 or 8 so that bigger proportion of work would be left at the parent node, thereby allowing for more processing nodes to access the work, instead of restricting the work to a smaller proportion of nodes.

More work is needed to study the effect of various minimum and maximum relation sizes on the ideal fragmentation size. A systematic study in this area might lead to heuristics or empirical formulas to identify the ideal fragmentation size a priori. The progress achieved at this front should have significant impact on the improvement of NOW-based systems since our data have consistently shown the performance sensitivity of NOW architectures to the fragmentation size.

This study has not considered the fault tolerance as a factor. In real NOW environment, however, node failure is fairly common. Therefore, this study can be extended into studying the job migration [25] and the performance of system in case of node failure.

References

- [1] A. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu and W. Zwaenepoel, "Treadmarks: shared memory computing on networks of workstations." *IEEE Computer* 29, pp.18-28, February, 1996.
- [2] T. Anderson, D. Culler and D. Patterson, "A case for NOW (networks of workstations)," *IEEE Micro* 15, pp. 54-64, February, 1995.
- [3] J. Arabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey and P. Stephan, "Dome: parallel programming in a heterogeneous multi-user environment", *Carnegie Mellon Univ. Sch. of Computer Sci. Tech Report CMU-CS-95-137 30786*, April 1995.
- [4] R. Arpaci-Dusseau, A. Arpaci-Dusseau, A. Vahdat, L. Liu, T. Anderson, and D. Patterson, "The interaction of parallel and sequential workloads on a network of workstations", *SIGMETRICS* 95, 1995.
- [5] A. Arpaci-Dusseau, R. Arpaci-Dusseau, and D. Culler, "Effective distributed scheduling of parallel workloads" *SIGMETRICS* 96-5, 1996.

- [6] A. Arpaci-Dusseau, and D. Culler, "Extending proportional-share scheduling to a network of workstations" *Proc. Int'l Conf. Parallel and Distributed Processing Techniques and Applications*, June 1997.
- [7] M. Atallah, C. Black and D. Marinescu, "Models and Algorithms for coscheduling compute-intensive tasks on a network of workstations". *Journal of Parallel and Distributed Computing*, 16, pp. 319-327, 1992.
- [8] R. Blumofe and D. Park, "Scheduling large-scale parallel computations on networks of workstations", *Proc. 3rd Int'l Symp on High-Performance Distr. Comput.*, San Francisco, CA, August 2-5, pp. 96-105, 1994.
- [9] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic and W. Su, "Myrinet: a gigabit-per-second local area network." *IEEE Micro* 15, pp. 29-36, February, 1995.
- [10] E. Brewer, F. Chong, L. Liu, J. Kubiawicz and S. Sharma, "Remote queues: exposing network queues for atomicity and optimization." *Proc. SPAA*, June, 1995.
- [11] J. Carter, "Design of the Munin distributed shared memory system", *Journal of Parallel and Distributed Computing*, 29, pp. 219-227, 1995.

- [12] J. Chen, and V. Li, "Optimizing Joins in Fragmented Database Systems on a Broadcast Local Network," *IEEE Trans. Software Engineering*, vol. 15, no. 1, pp. 26-38, 1989.
- [13] M. Chen, P. Yu, and K. Wu, "Optimization of Parallel Execution for Multi-join Queries," *IEEE Trans. Knowledge and Data Engineering*, vol. 8, no. 3, pp. 416-428, 1996.
- [14] S. Chung, and J. Yang, "A Parallel Distributive Join Algorithm for CUBE-connected Multiprocessors," *IEEE Trans. Parallel and Distributed Systems*, vol. 7, no. 2, pp. 127-138, 1996.
- [15] M. Crovella, P. Das, C. Dubnicki, T. LeBlanc, and E. Markatos, "Multiprogramming on multiprocessors", *Proc. 3rd IEEE Symp. on Parallel and Distributed Processing*, pp. 590-597, December, 1991.
- [16] D. Culler, L T. Liu, R. Martin, C. Yoshikawa, "LogP performance assessment of fast network interfaces", *IEEE Micro*, 16, pp.35-43, February, 1996.
- [17] D. Culler, A. Arpaci-Dusseau, R. Arpaci-Dusseau, B. Chun, S. Lumetta, A. Mainwaring, R. Martin, C. Yoshikawa and F. Wong, "Parallel computing on the

Berkeley NOW", *JSPP'97(9th Joint Symposium on Parallel Processing)*, Kobe, Japan, 1997.

[18] S. Dandamudi and S. P. Cheng, "A hierarchical task queue organization for shared-memory multiprocessor system", *IEEE Trans. on Parallel and Distributed Systems*, 6(1), pp. 1-16, January 1995.

[19] S. Dandamudi, "Using Networks of Workstations for Database Query Operations," *Int'l Conf. Computer and Their Applications*, 1997.

[20] S. Dandamudi and G. Jain, "Architectures for parallel query processing on networks of workstations", *Int. Conf. Parallel and Distributed Computing Systems*, pp. 444-451, New Orleans, October, 1997.

[21] S. Dandamudi and T. K. Thyagaraj, "A hierarchical processor scheduling policy for distributed-memory multicomputer systems", *IEEE Proceedings of the 4th International Conference on High Performance Computing*, pp. 218-223, 1997.

[22] R. Dannenberg and P. Hibbard, "A butler process for resource sharing on spice machines", *ACM Trans. on Office Information Systems*, 3(3), pp.235-252, 1985.

- [23] M. de Prycker, "Asynchronous Transfer Mode: solution for broadband ISDN", *Ellis Horwood Publishers, 2nd ed. 1993.*
- [24] D. Dewitt, and J. Gray, "Parallel Database Systems: the Future of High Performance Database Systems," *Communications of ACM*, vol. 35, no. 6, pp. 85-98, 1992.
- [25] F. Douglass and J. Ousterhout, "Transparent process migration: design alternatives and the Sprite implementation", *Software-Practice and Experience*, 21(8), pp. 757-785, August, 1991.
- [26] D. Eager and J. Zahorjan, "Adaptive guided self-scheduling", *Univ. of Washington, Tech Report 92-01-01*, January, 1992.
- [27] K. Efe and V. Krishnamoorthy, "Optimal scheduling of compute-intensive tasks on a network of workstations", *IEEE Trans. Parallel and Distributed Systems*, 6(6), pp. 668-673, 1995.
- [28] T. von Eicken, D. Culler, S. Goldstein and K. Schauser, "Active messages: a mechanism for integrated communication and computation. *Proc. 19th Int'l Symp. on Computer Architecture (also available UC Berkeley Tech Report CSD 92/675)*, May 1992.

- [29] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser, "Active messages: a mechanism for integrated communication and computation", *Proc. 9th ISCA, Gold Coast, Australia*, May, 1992.
- [30] T. von Eicken, A. Basu and V. Buch, "Low-latency communication over ATM networks using Active Messages." *IEEE Micro* 15, pp. 54-64, February, 1995.
- [31] T. von Eicken, A. Basu, V. Buch, and W. Vogel, "U-Net: a user-level network interface for parallel and distributed computing", *Proc. 15th SOSP*, pp. 40-53, December, 1995b.
- [32] D. Feitelson and L. Rudolph. "Gang scheduling performance benefits for fine-grained synchronization", *Journal of Parallel and Distributed Computing*, 16(4), pp. 306-318, December, 1992.
- [33] R. Hagmann, "Process server: sharing processor power in a workstation environment", *Proc. of 6th Int'l Conf. on Distributed Computing*, pp.260-267. May, 1986.
- [34] K. Hwang, W. Croft, "A Unix-based local computer network with load balancing", *IEEE Computer*, 15 (4), pp.55-66, April, 1982.

- [35] M. Karlsson and P. Stenstrom, "Performance evaluation of a cluster based multiprocessor built from ATM switches and bus-based multiprocessor servers", *Chalmer Univ. of Technology, Dept. Computer Sci Tech Report S-412*, 1996.
- [36] K. Keeton, T. Anderson, and D. Patterson, "LogP quantified: the case for low-overhead local area networks", *Hot Interconnects III*, August, 1995.
- [37] E. Kemal and V. Krishnamoorthy, "Optimal scheduling of computer-intensive tasks on a network of workstations," *IEEE Trans Parallel and Distributed Syst.* 6, pp. 668-673, June, 1995.
- [38] P. Krueger and R. Chawla, "The stealth distributed scheduler," *Proc. 11th Conf. Distrib. Comput. Syst.*, pp. 336-343, 1991.
- [39] C. Kruskal and J. Weiss, "Allocating independent tasks on parallel processors", *IEEE Trans. Software Engineering*, 11, pp. 1001-1016, October, 1995.
- [40] E. Lazowska, D. Eager and J. Zahorjan, "Adaptive load sharing in homogeneous distributed systems", *IEEE Trans. Software Engineering*, 12, pp. 662-675, May, 1986.

- [41] E. Lazowska, D. Eager and J. Zahorjan, "The limited performance benefits of migrating active processes for load sharing", *Performance Evaluation Review, ACM*, 16, pp. 63-72, May, 1988.
- [42] J. Liebeherr, E. Omiecinski, and I. Akyildiz, "The Effect of Index Partitioning Schemes on the Performance of Distributed Query Processing," *IEEE Trans. Knowledge and Data Engineering*, vol. 5, no. 3, pp. 510-522, 1993.
- [43] F. Lin and R. Keller, "The gradient model load balancing method", *IEEE Trans. Software Engineering*, SE-13, pp. 32-38, January, 1987.
- [44] Liu et. al., "Scheduling parallel loops with variable length iteration execution times on parallel computers", *Proc. 5th Int'l Conf. Parallel and Distributed Computing and System*, October, 1992.
- [45] S. Lucco, "A dynamic scheduling method for irregular parallel programs", *ACM SIGPLAN'92 Conf. Programming Language Design and Implementation*, pp. 200-211, 1992.
- [46] E. Markatos and T. LeBlanc, "Using processor affinity in loop scheduling on shared memory multiprocessors", *IEEE Trans. Parallel Distributed Systems*, 5(4), April, 1994.

- [47] R. Martin, "An active message layer for a network of workstations", *Proc. 2nd Hot Interconnects Conf*, July, 1994.
- [48] R. Martin, A. Vahdat, D. Culler and T. Anderson, "Effect of communication latency, overhead, and bandwidth on a cluster architecture," *Int'l Symp. on Computer Architecture*, June, 1997.
- [49] Message Passing Interface Forum, "MPI: a message-passing interface standard, version 1.0", May, 1994.
- [50] N. Nedeljkovic and M. Quinn, "Data parallel programming on a network of heterogeneous workstations", *Proc. of the 1st Int'l Symp. on High Performance Distributed Computing*, pp. 152-166, September, 1992.
- [51] J. Ousterhout, "Scheduling techniques for concurrent systems", *Proc. 3rd Int'l Conf. on Distributed Computing Systems*, pp. 22-30, May, 1982.
- [52] H. Pirahesh, C. Mohan, J. Cheng, T. Liu, and P. Sellinger, "Parallelism in Relational Data Base Systems: Architectural Issues and Design Approaches," *2nd Int'l Symp. Databases in Parallel and Distributed Systems*, pp. 4-29, 1990.

- [53] C. Polychronopoulos and D. Kuck, "Guided self-scheduling: a practical scheduling scheme for parallel supercomputers", *IEEE Trans. Computer*, C-36(12), pp. 1425-1439, December, 1987.
- [54] J. Richardson, H. Lu, and K. Mikkilineni, "Design and Evaluation of Parallel Pipelined Join Algorithms," *Proc. ACM SIGMOD*, pp. 399-409, 1987.
- [55] S. Rodrigues, T. Anderson, and D. Culler, "High-performance local area communication with fast socket", *USENIX'97*, 1997.
- [56] V. Saletore, J. Jacob, and M. Padala, "Parallel computation on a network of heterogeneous workstations", *Proc. 3rd IEEE Int'l Symp. on High Performance Distr. Computing*, April, 1994.
- [57] D. Schneider, and D. Dewitt, "A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment." *SIGMOD Record* (ACM Special Interest Group on Management of Data), vol. 18, no. 2, pp. 110-121, 1989.
- [58] B. S. Siegell, "Automatic generation of parallel programs with dynamic load balancing for a network of workstations, *Carnegie Mellon Univ. Sch. of Computer Sci. CMU-CS-95-168 30880*, May, 1995.

- [59] P. Sobalvarro, S. Pakin, W. Weihl, and A. Chien, "Dynamic coscheduling on workstation clusters", *<http://www.research.digital.com/SRC/scheduling>*, 1997.
- [60] C. Soleimany, "Distributed Parallel Query Processing on Network of Workstations", *M.C.S. thesis*, 1998.
- [61] V. Sunderam, "PVM: a framework for parallel distributed computing". *Concurrency: Practice and Experience*, 2(4), pp. 315-339, December, 1990.
- [62] V. Sunderam, "The PVM system: Status, Trends, and Directions." *Lecture Notes in Computer Science (Enro PVM'96)*, pp. 68-80, 1996.
- [63] P. Tang and P. Yew, "Processor self-scheduling for multiple nested parallel loops". *Proc. '86 Int'l Conf on Parallel Processing*, August, 1986.
- [64] M. Theimer and K. Lantz, "Finding idle machines in a workstation-based distributed system", *IEEE Trans. Software Eng.* 15 (11), pp. 1444-1457, 1989.
- [65] T. Tzen and L. Ni, "Trapezoid self-scheduling: a practical scheduling scheme for parallel computers", *IEEE Trans. Parallel Distributed Systems*, 4, pp.87-98, January, 1993.

- [66] P. Valduriez, and G. Gardarin, "Join and Semijoin Algorithms for a Multiprocessor Database Machine," *ACM Trans. Database Systems*, vol. 9, no. 1, pp. 133-161, 1984.
- [67] X. Wang, and W. Luk, "Parallel Join Algorithm on a Network of Workstations," *Int'l Symp. Database in Parallel and Distributed Systems*, pp. 87-95, 1988.
- [68] Y. Yan, C. Jin, and X. Zhang, "Adaptively scheduling parallel loops in distributed shared-memory systems", *IEEE Trans. on Parallel and Distributed Systems*, 8(3), pp. 70-81, January, 1997.
- [69] M. Zaki, W. Li and S. Parthasarathy, "Customized dynamic load balancing for a network of workstations," *Univ. Rochester Tech Report 602*, December, 1995.